Can Languages and Tools Produce Better Software?

- In this talk, better means more reliable
  - Less likely to fail

SPT Approach

- Redundency is good
- Redundancy exposes inconsistency

SPT Approach

- Redundency is good
- Redundancy exposes inconsistency
- Inconsistency points to errors

SPT Approach

- Redundency is good
- Redundancy exposes inconsistency
- Inconsistency points to errors
- Compare what programmer should do against what code does

Lightweight Specifications

- Start with description of correct/incorrect behavior
  - Not total specification ⇒ not total correctness
- Programmers provide specs
- Tools reward additional effort
  - Mechanically and systematically find errors

Programmers Write Types

- Successful, wide accepted specifications
  - Programmers understand and write type declarations
  - Declarations aid comprehension
  - Type checking is fast, routine, and finds errors
- Serve several roles
  - Documentation of interface syntax
  - Basis for program abstractions
  - Static (compile-time) error detection

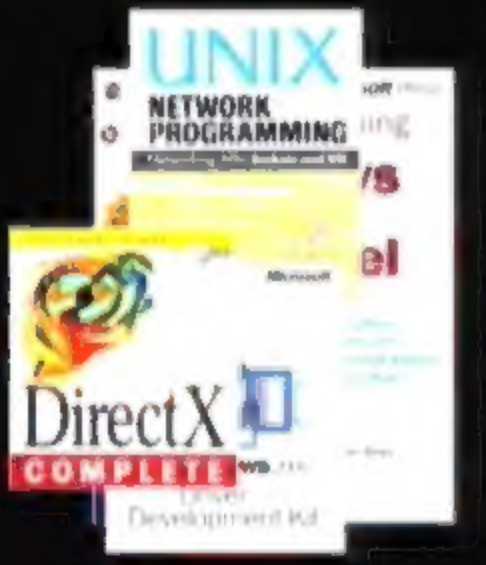
Extend Successful Approach

- Specify and check other program properties
  - Languages (programming and domain-specific) express properties
  - Tools check that code obeys rules
- Goal is partial correctness
  - Detect and report important classes of errors
  - No guarantee of program correctness
- Systematic
  - Sound, complete analysis can flag absence of errors

SPT Overview

- Reliable interfaces
  - Vault
  - SLAM
  - ESP
- Concurrency and security
  - Behave!
  - Golf

Interfaces Have Usage Rules



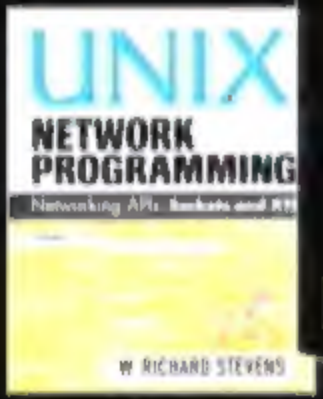
- Specs in documentation
  - Incomplete, unenforced, wordy
  - Order of operations & data access
  - Resource management
- Disobeying rules causes bad behavior
  - System crash or deadlock
  - Unexpected exceptions
  - Failed runtime checks
  - Security flaws

Example Usage Rule: Sockets

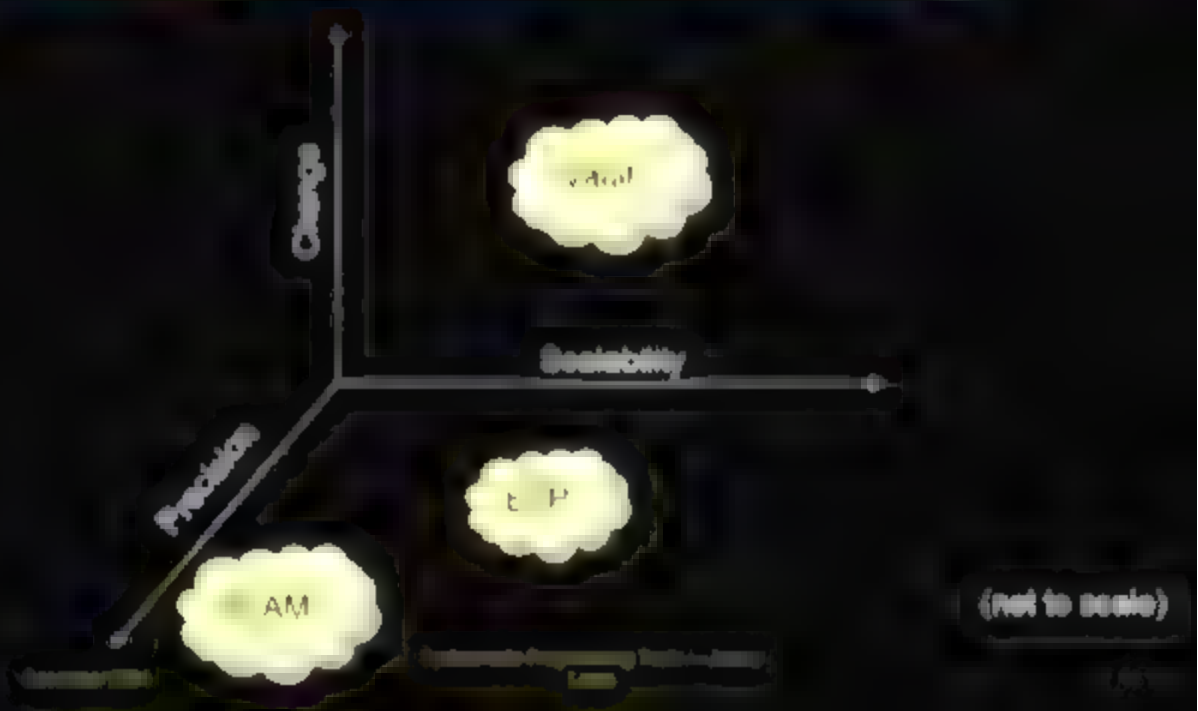
the "communication domain" in which communication is to take place; see protocols(5).

Sockets of type SOCK_STREAM are full-duplex byte streams, similar to pipes. A stream socket must be in a connected state before any data may be sent or received on it. A connection to another socket is created with a connect(2) call. Once connected, data may be transferred using read(2V) and write(2V) calls or some variant of the send(2) and recv(2) calls. When a session has been completed a close(2V), may be performed. Out-of-band data may also be transmitted as described in send(2) and received as described in recv(2).

The communications protocols used to implement a SOCK_STREAM insure that data is not lost or duplicated. If a piece of



SPT Specification & Checking



SPT Overview

- Reliable interfaces
  - Vault
  - SLAM
  - ESP
- Security and concurrency
  - Behavel
  - Golf

Vault (Rob DeLine, Manuel Fähndrich)

- Vault folds usage rules into programming language's type system
  - Interface author states rules in type signature
  - Compiler rejects client code that violates rule (type error)
  - Every violation guaranteed to be found (soundness)

Why a Type-based Approach?

- Types are widely accepted form of specification
- Specification and code are kept in sync
- Developers find and fix bugs before execution
- Programmers understand violations and how to fix them
- Efficient, scalable analysis doesn't slow development

Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int)   [ S @ named => listening ]
```

Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int)   [ S @ named => listening ]
```

Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int)   [ S @ named -> listening ]
```

- Naming: tracked types give compile-time names (keys) to objects

Tracked Types and Keys

- Usage rules talk about individual objects

```
void listen (tracked(S) sock, int)  [ S @ named -> listening ]
```

- Naming: tracked types give compile-time names (keys) to objects
- Object state: key can have symbolic state to model state of object

Tracked Types and Keys

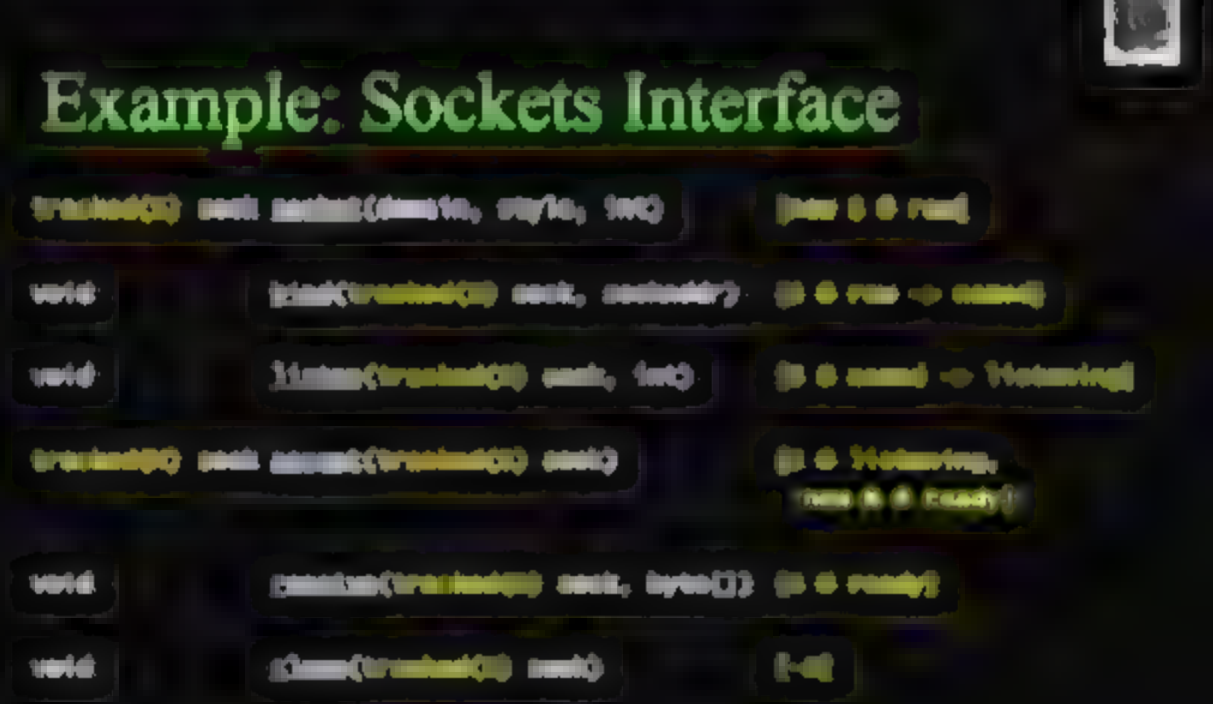
- Change specs combine pre/post condition for functions

Tracked Types and Keys

- Change specs combine pre/post condition for functions
- Add key:
  - `tracked(s) sock socket (...) [ new s @ raw ]`
- Consume key:
  - `void close (tracked(s) sock) [ -s ]`
- Change key state:
  - `void bind (tracked(s) sock) [ s @ raw -> named]`

Example: Sockets Interface

Example: Sockets Interface



Statically Enforcing Usage Protocols

- At every program point, Vault compiler computes key set
  - Functions and built-in operations (new/free) change key set
  - Key must be consumed, not "leaked"
- On each path in function's body, check:
  - Pre-condition transformed into post-condition
  - All proof obligations satisfied
    - Pre-conditions of other function calls
    - Primitive operations (memory access, free)
- Avoid exponential blow-up
  - Require uniform predicate at join points

Checking Socket Client Code

```
void communicate(sockaddr addr) {
  [illegible] sock s = socket('[illegible]', '[illegible]', 0);
  bind(s, addr);
  listen(s, 0);
  while (![illegible]()) {
    [illegible] sock c = accept(s);
    receive(s, buffer);
    close(c);
  }
  close(s);
}
```

Checking Socket Client Code

Checking Socket Client Code

```
void communicate(sockaddr addr) {
  [redacted] sock = socket(AF_INET, SOCK_DGRAM, 0);
  bind(s, addr);
  listen(s, 0);
  while (!shutdown()) {
    [redacted] sock c = accept(s);
    receive(s, buffer);
    close[redacted];
  }
  close(s);
}
```

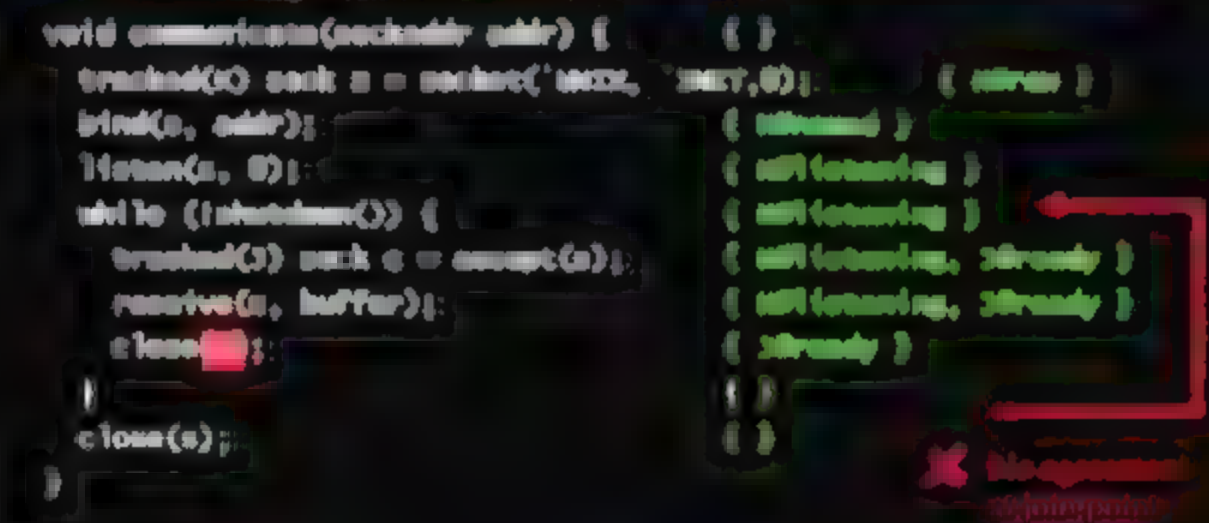
Checking Socket Client Code

Checking (Wrong) Socket Client Code

```
void communicate(sockaddr addr) {
  int s = socket(AF_INET, SOCK_DGRAM, 0);
  bind(s, addr);
  listen(s, 0);
  while (!shutdown()) {
    int c = accept(s);
    receive(s, buffer);
    close(c);
  }
  close(s);
}
```

Checking (Wrong) Socket Client Code

Checking (Wrong) Socket Client Code



Checking (Wrong) Socket Client Code

```
void communicate(sockaddr addr) {
  int s = socket(AF_INET, SOCK_STREAM, 0);
  bind(s, addr);
  listen(s, 0);
  while (!shutdown()) {
    int c = accept(s);
    receive(s, buffer);
    close(c);
  }
  close(s);
}
```

```
{ }
{ AF_INET }
{ SOCK_STREAM }
{ addr }
{ s, addr }
{ s, 0 }
{ s, 0 }
{ s, shutdown }
{ s, shutdown }
{ shutdown }
{ }
{ }
```

✗ Looking buy T

Checking (Wrong) Socket Client Code

Vault Research

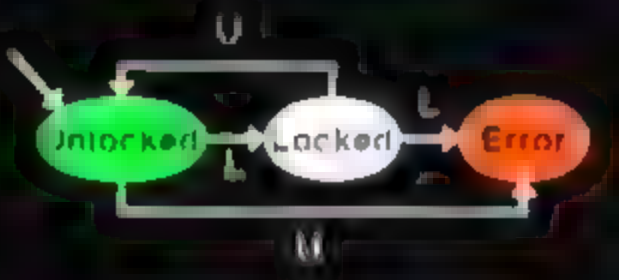
- Few new language features express and check many usage rules
  - Still imperative programming style
  - Handle "real" applications: device drivers, DirectX
  - Check "real" rules that developers commonly violate
- On-going research
  - Usage rules in C#
  - Safe, explicit memory management in CLR

SLAM (Tom Ball, Sriram Rajamani)

- Input
  - C source code, "as is"
  - API rules in SLIC language
- Automatically create abstraction of C program
  - Abstract model = Boolean program
- Systematic exploration of model's state space
  - Does feasible path through program lead to error state in SLIC?
- Demand-driven refinement of model
  - Exclude infeasible paths

API Rule for Locking

State Machine



API Rule for Locking

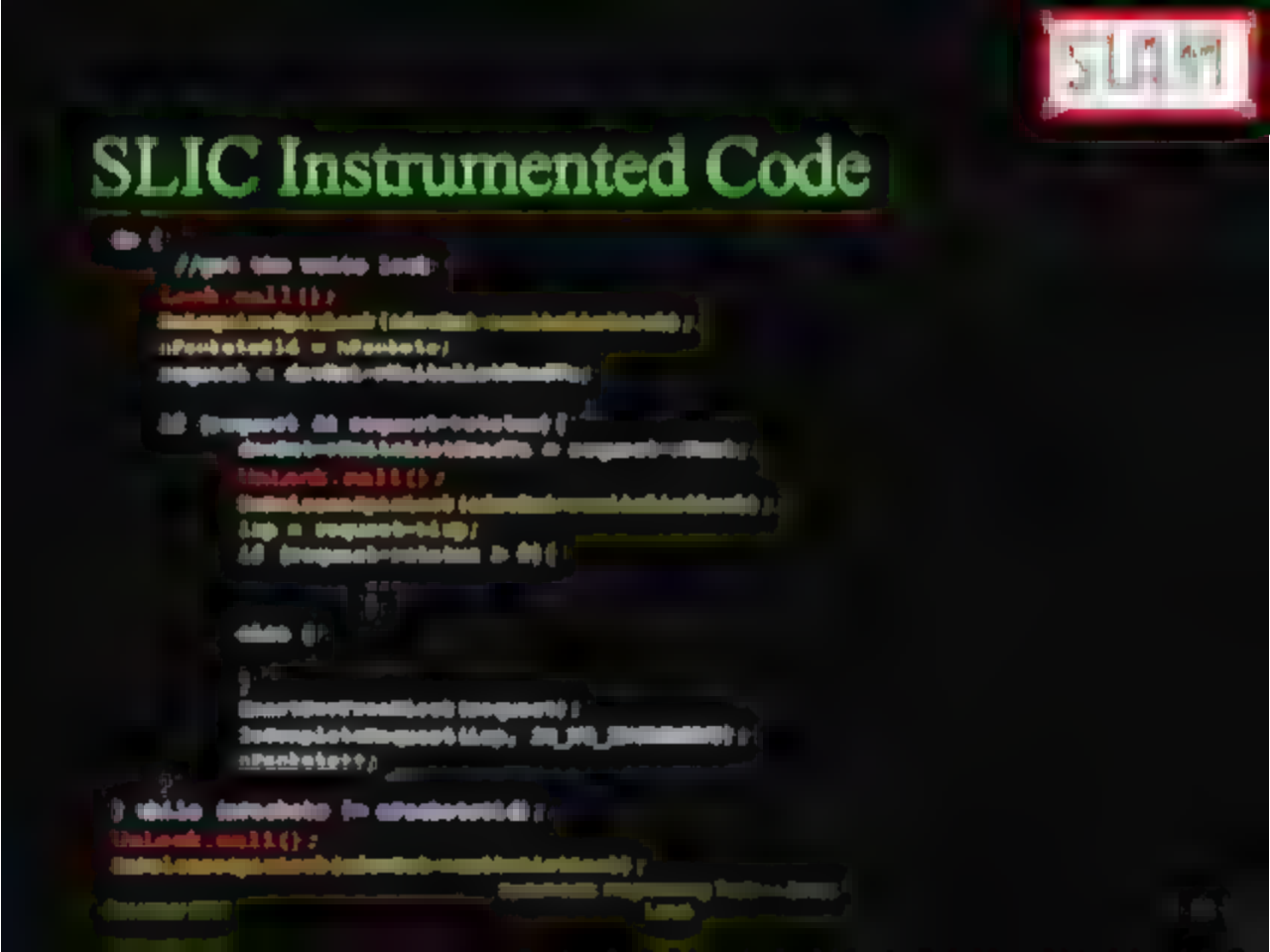
State Machine

SLIC

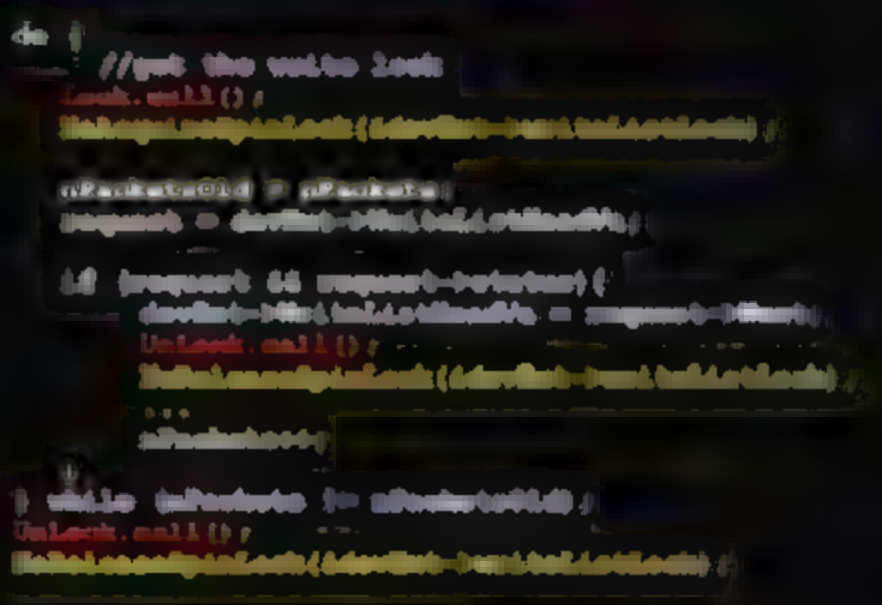
```
state {
  int locked = 0;
}

Lock.call {
  if (locked==1) abort;
  else locked = 1;
}

Unlock.call {
  if (locked==0) abort;
  else locked = 0;
}
```

Is Locking Protocol Respected?

SLIC Instrumented Code



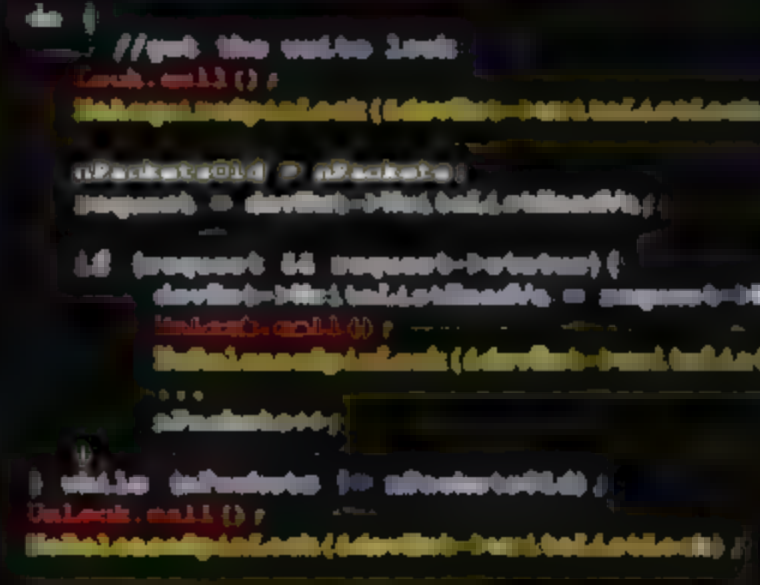
State-based Search Fails



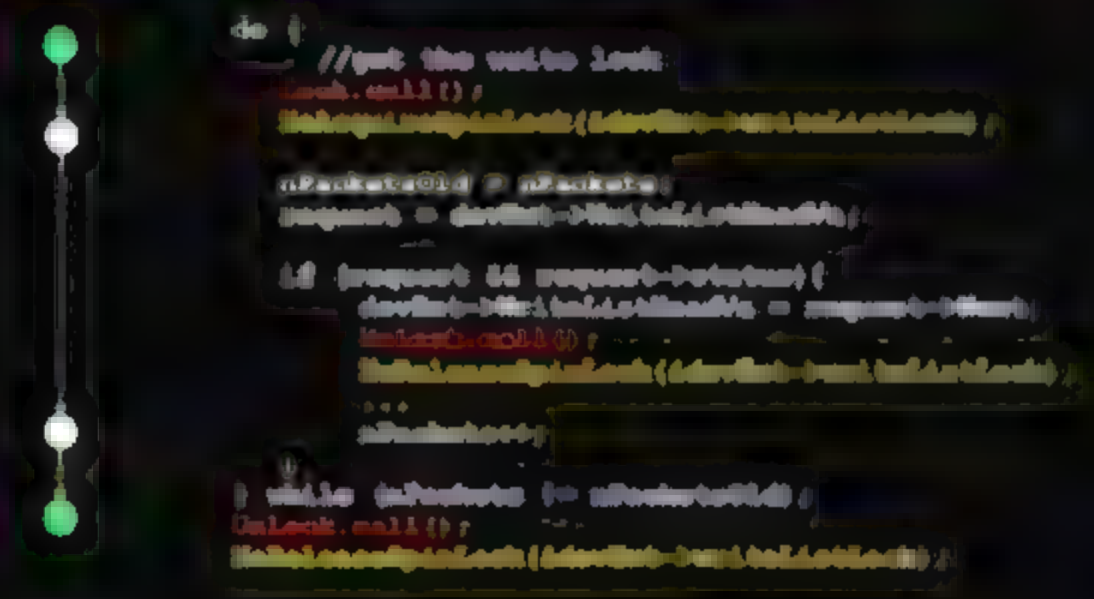
State-based Search Fails

State-based Search Fails

State-based Search Fails

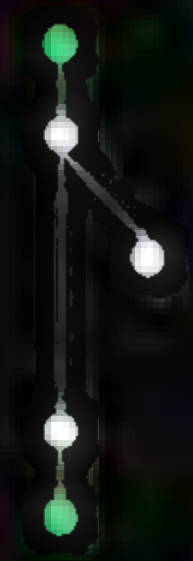


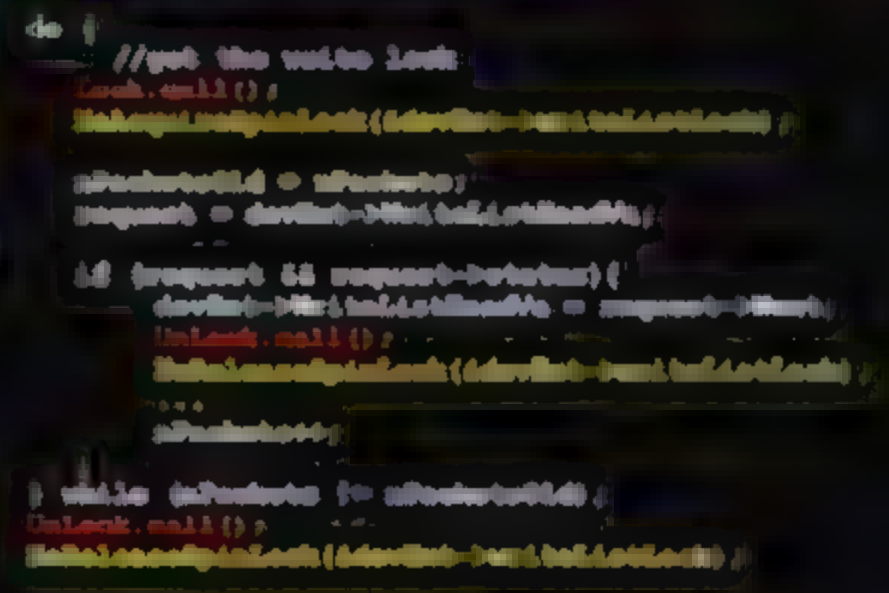
State-based Search Fails



State-based Search Fails

State-based Search Fails





State-based Search Fails

```
do {
    //get the value level
    Level.val();
    [obj getObjectAtLevel:(level+1) withValue:&value];

    nPacketsOld = nPackets;
    request = [self findRequest:&value];

    if (request && request->active){
        [self.tableView beginUpdates];
        [tableView deleteRowsAtIndexPaths:
        ...
        [tableView endUpdates];
    }
} while (nPackets != nPacketsOld);
[level.val();
[obj getObjectAtLevel:(level+1) withValue:&value];
```

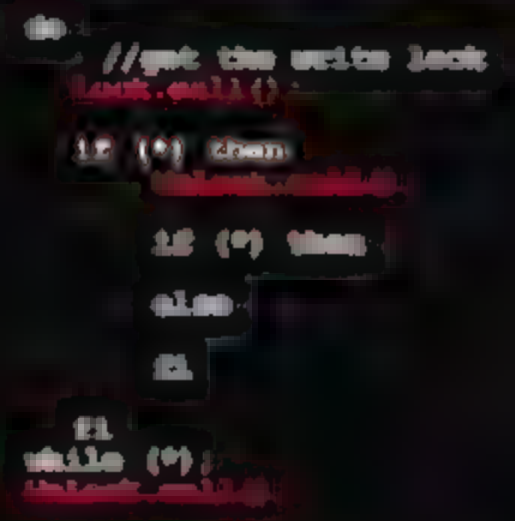
State-based Search Fails

```
do {
    //get the write lock
    lock.wait();
    ...
    nPacketsOld = pPackets;
    request = devMgr->findLowPriorityReq();
    ...
    if (request && request->active) {
        devMgr->freeLowPriorityReq = request->next;
        Unlock.call();
        ...
        nPacketsOld++;
    }
} while (nPackets != nPacketsOld);
Unlock.call();
```

State-based Search Fails

```
do {
    //get the write lock
    lock.wait();
    ...
    if (request && request->active) {
        ...
        unlock.wait();
        ...
    }
} while (...);
unlock.wait();
```

Boolean Program; Abstraction 1



Boolean Program; Abstraction 1

```
do
    //get the write lock
    lock.call();
    if (*) then
        unlock.call();
        if (*) then
        else
        fi
    fi
od;
while (*);
```

Boolean Program; Abstraction 1

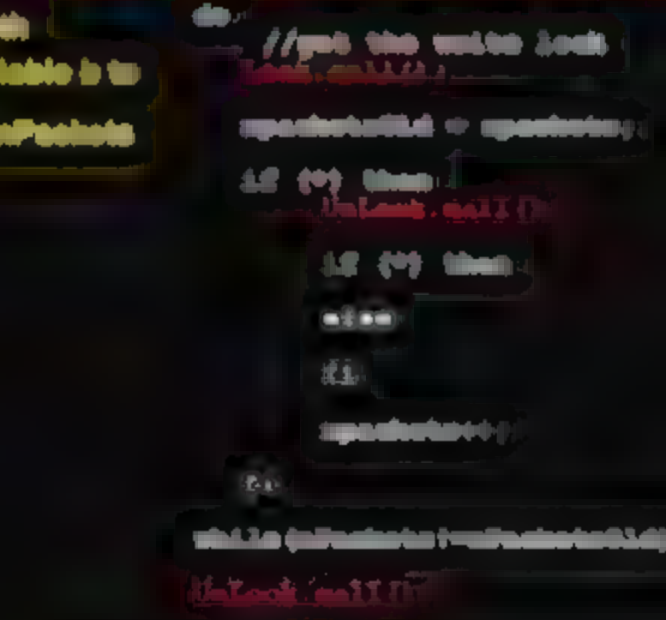
```
do
    //get the write lock
    lock.call();
    if (*) then
        unlock.call();
        if (*) then
        else
        fi
    fi
od
```

Boolean Program; Abstraction 1

```
do {
    //get the write lock
    lock.call();
    if (*) then
        unlock.call();
        if (*) then
        else
        fi
    fi
} while (*);
```

Boolean Program; Refine Abstraction

- Examine infeasible path
- Introduce boolean variable b to represent:
  nPacketsOld == nPackets



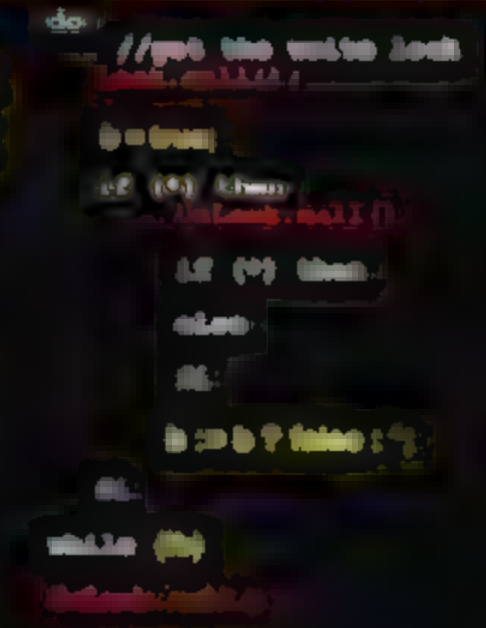
Boolean Program; Refine Abstraction

- [illegible infeasible path]
- [introduces boolean variable b to represent:
  nPacketsOld == nPackets]

```
do
  //get the write lock
  KeAcquireSpinLock(...);
  nPacketsOld = nPackets;
  if (*) then
    IoReleaseCancelSpinLock(...);
    if (*) then
    else
    fi;
    nPackets++;
  fi;
while (nPacketsOld != nPackets);
IoReleaseCancelSpinLock(...);
```

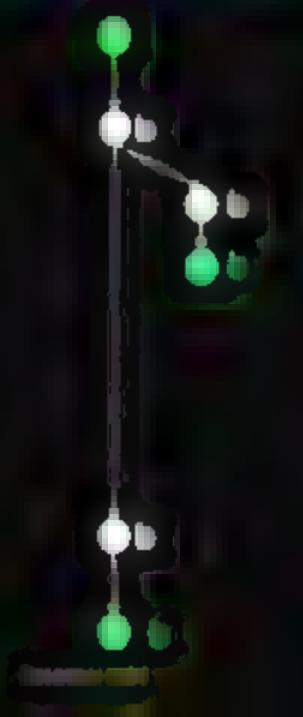
Boolean Program; Refine Abstraction

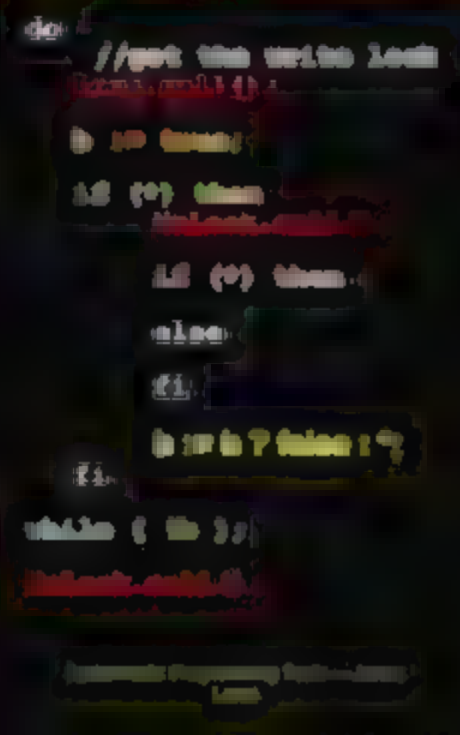
- Learning infeasible path
- Introduce boolean variable b to represent:
  nPacketsOld == nPackets



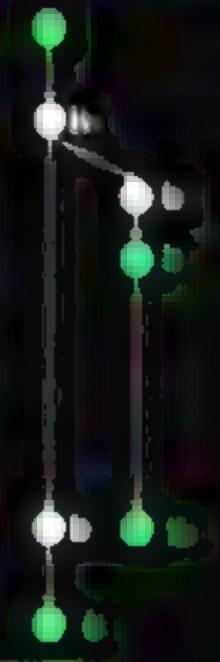
Boolean Program; Abstraction 2

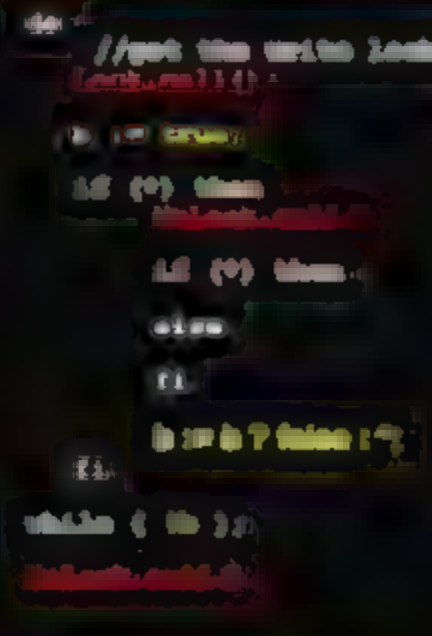
Boolean Program; Abstraction 2



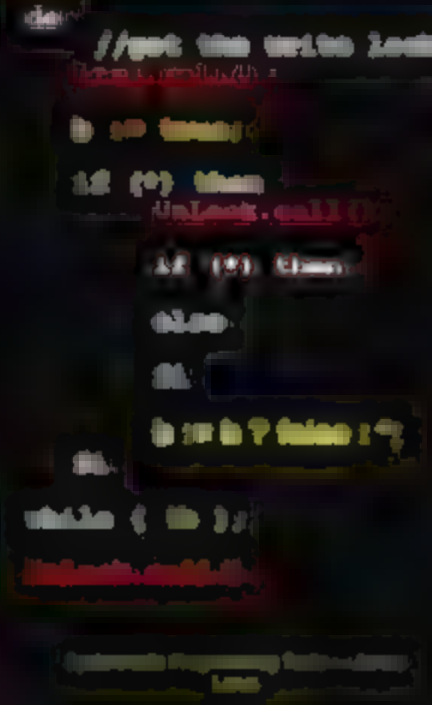


Boolean Program; Abstraction 2

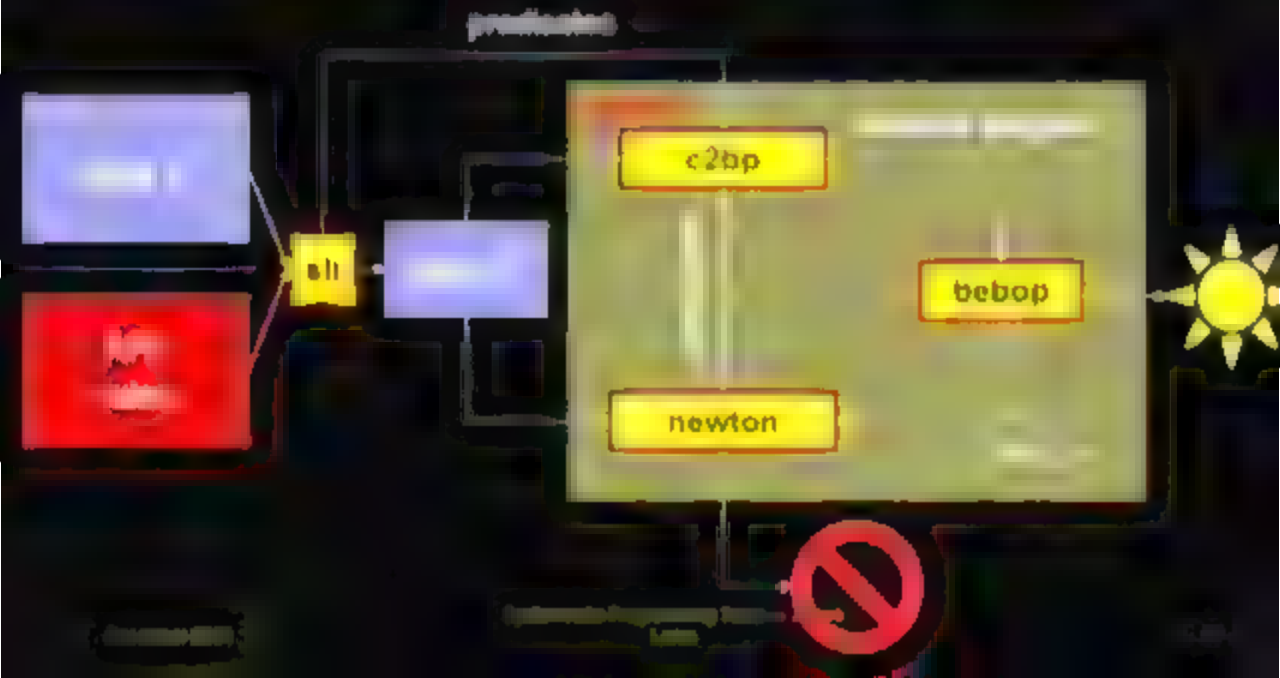




Boolean Program; Abstraction 2



Checking API Rules with SLAM



SLAM Summary

- Automatic analysis (no code annotation)
  - SLIC specification of API rules
- Report errors with source-level paths
- Very precise analysis
  - Minimize false positives
- Current research
  - Applying to device drivers
  - Tool scalability

The Static Driver Verifier

Thomas Ball
Sriram K. Rajamani

Software Productivity Tools

Programmer Productivity
Research Center

The Static Driver Verifier

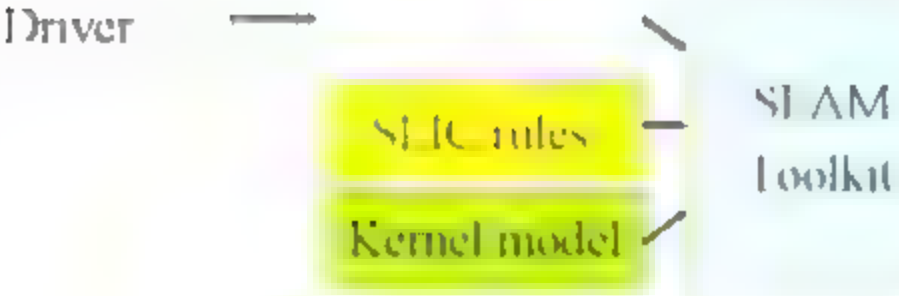
Thomas Ball
Sriram K. Rajamani

Software Productivity Tools

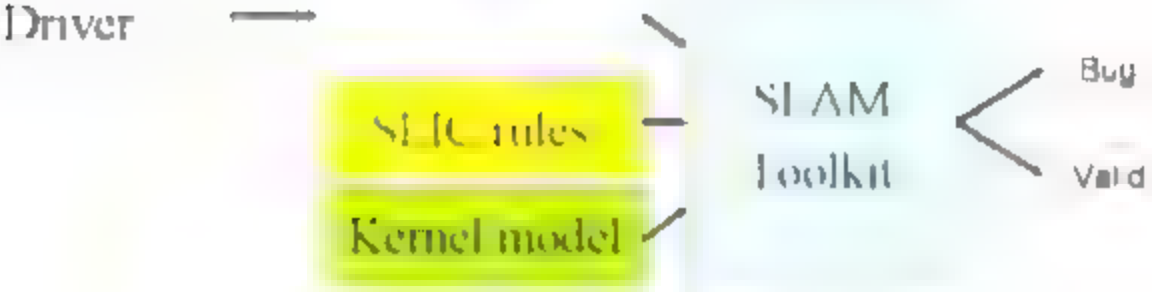
Programmer Productivity
Research Center

Static Driver Verifier

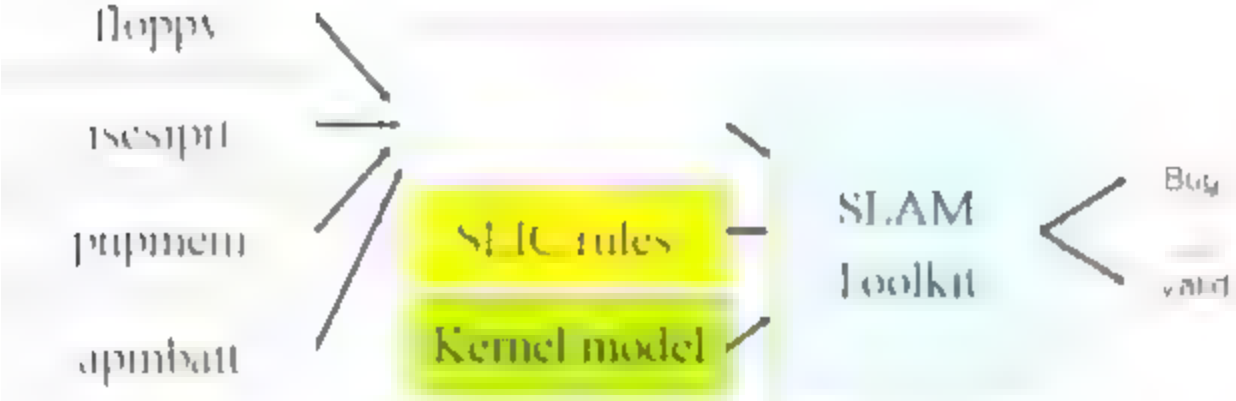
Static Driver Verifier

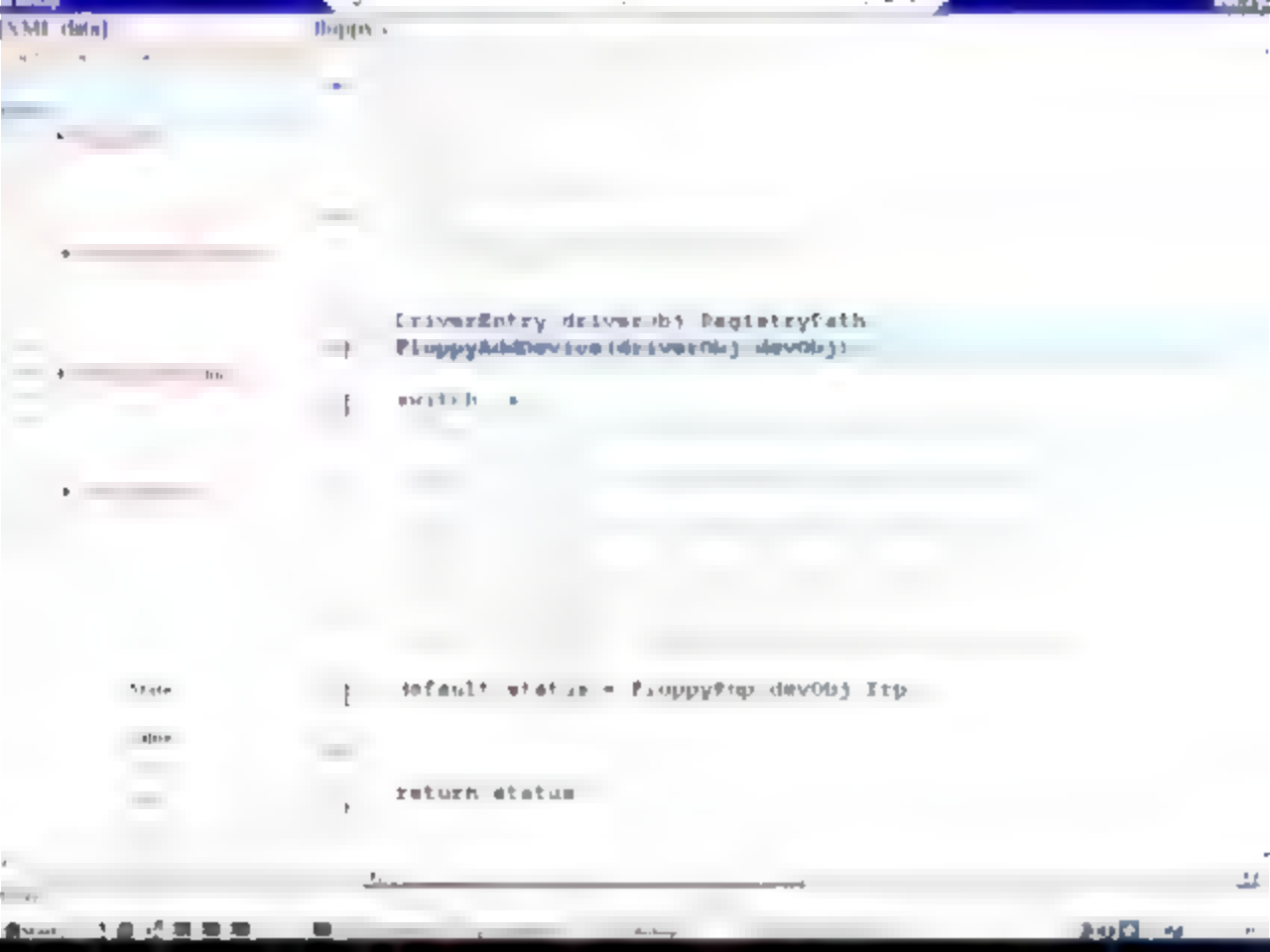


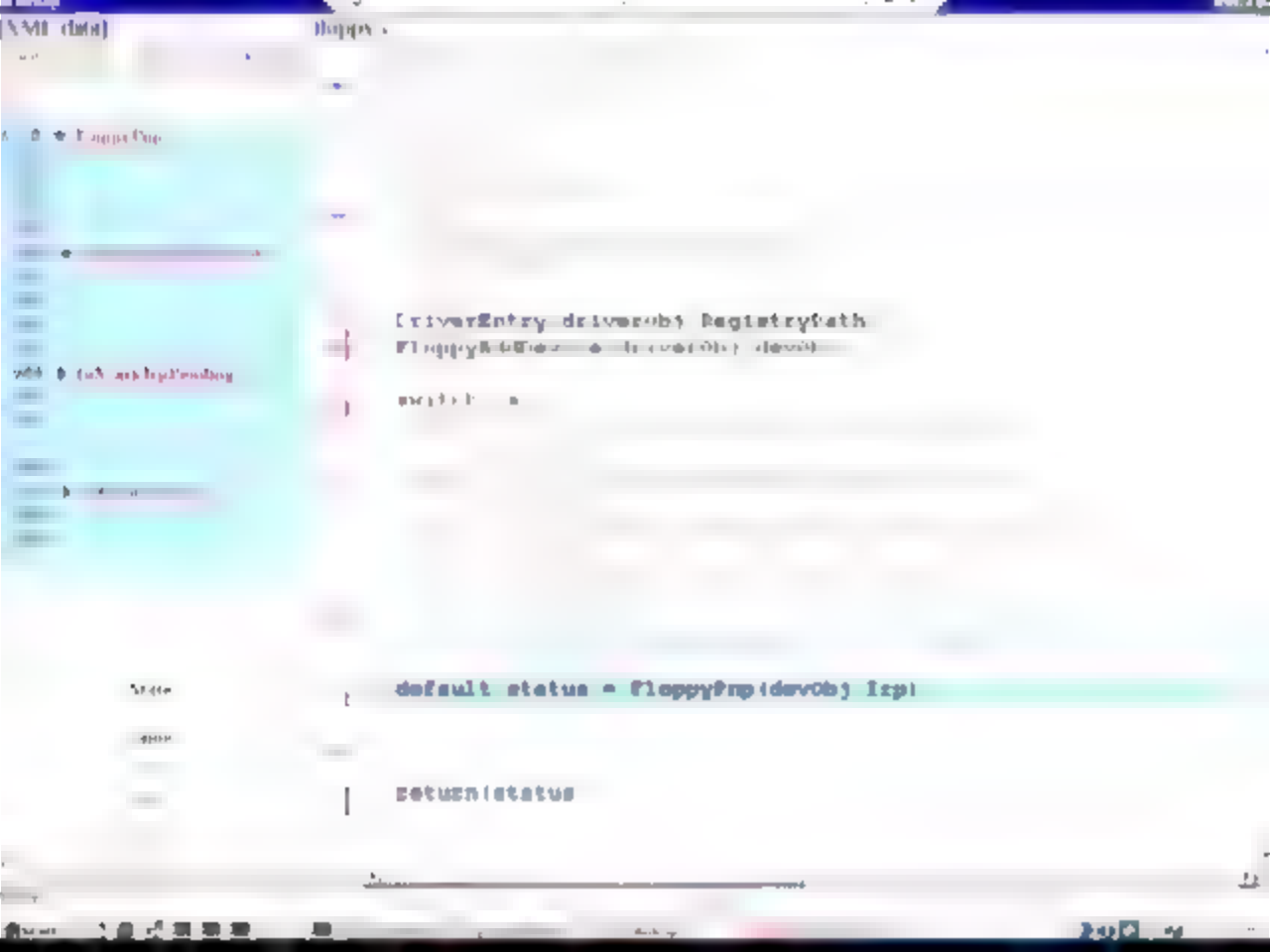
Static Driver Verifier

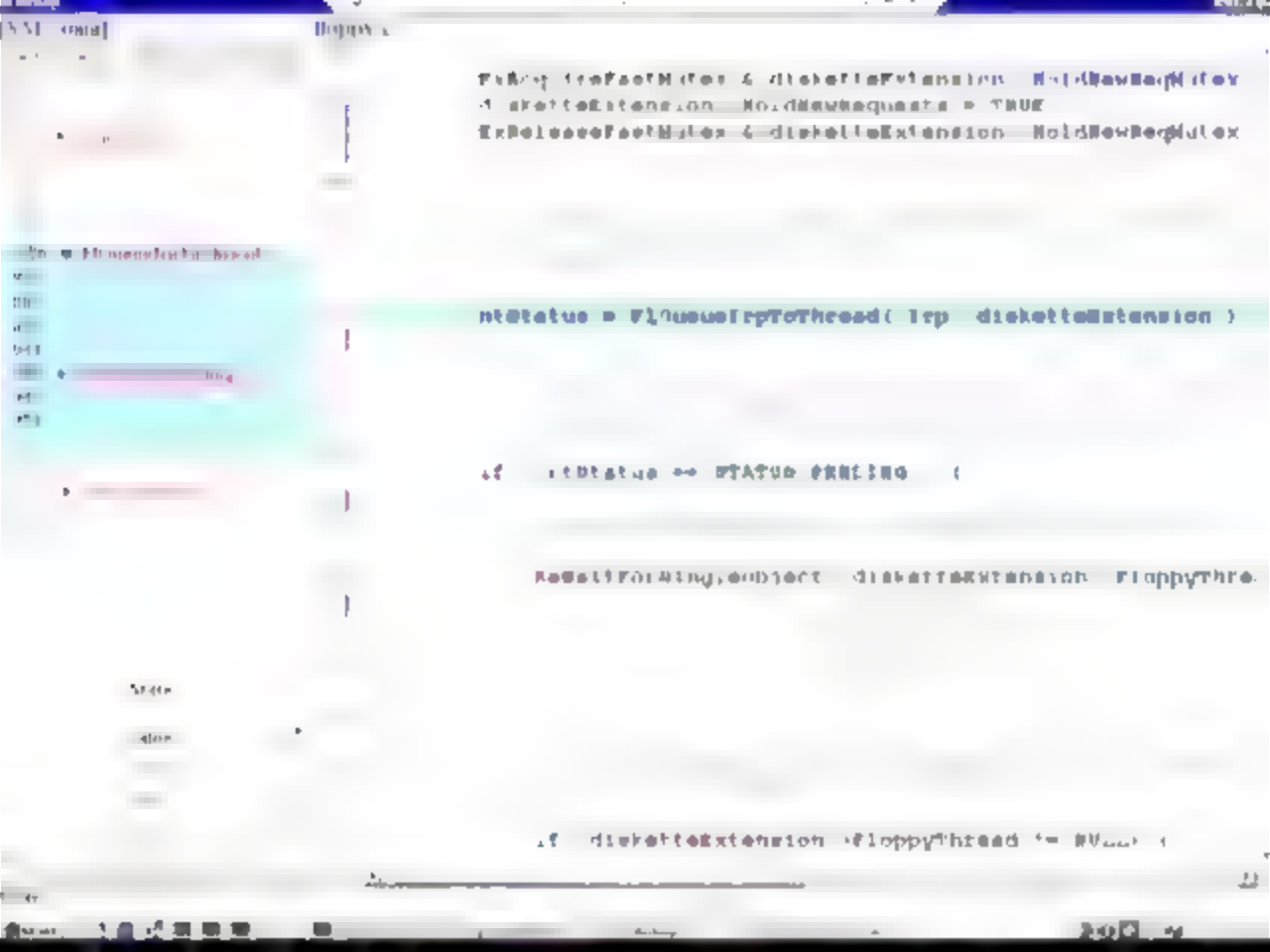


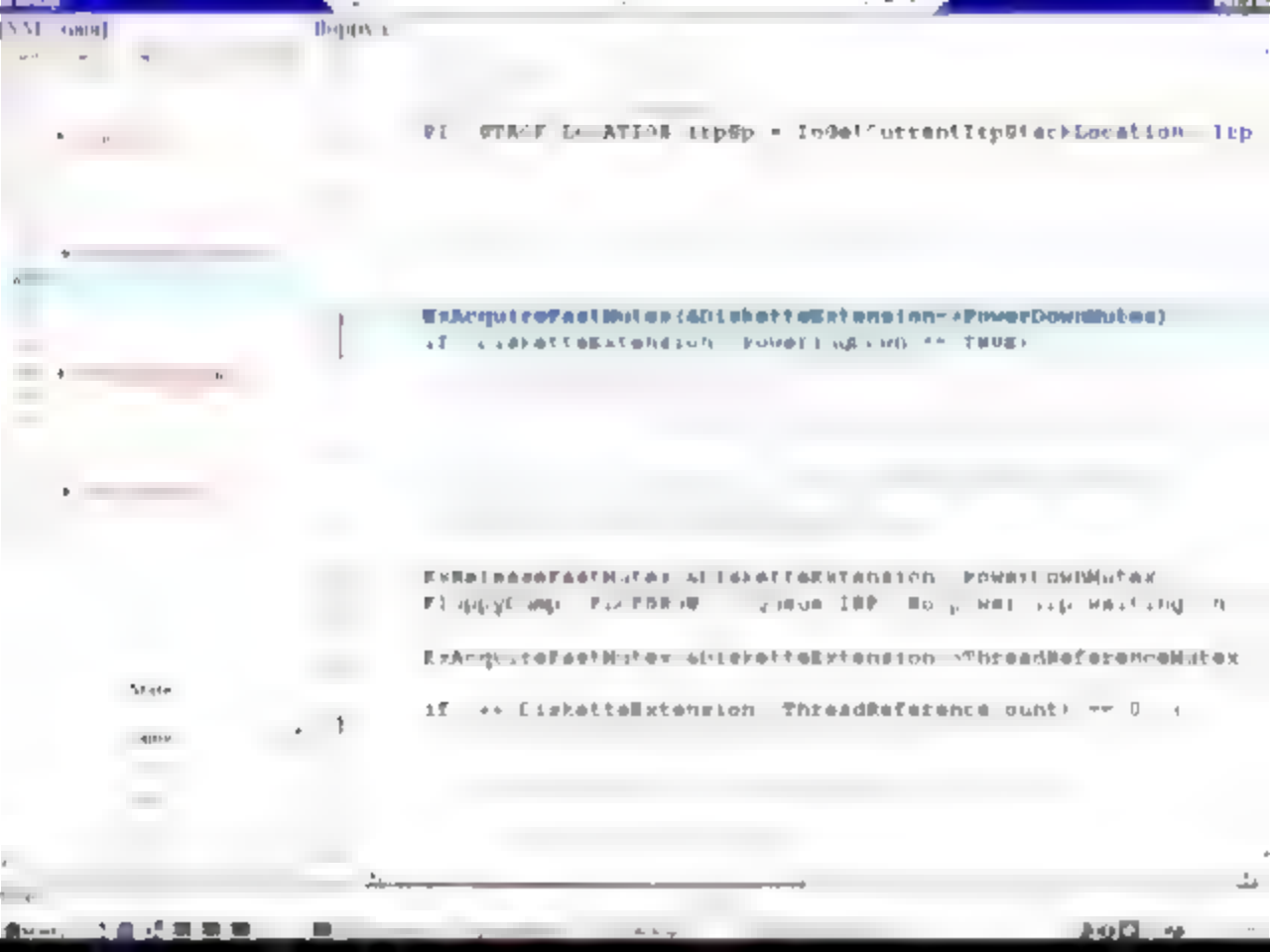
Static Driver Verifier (ii)

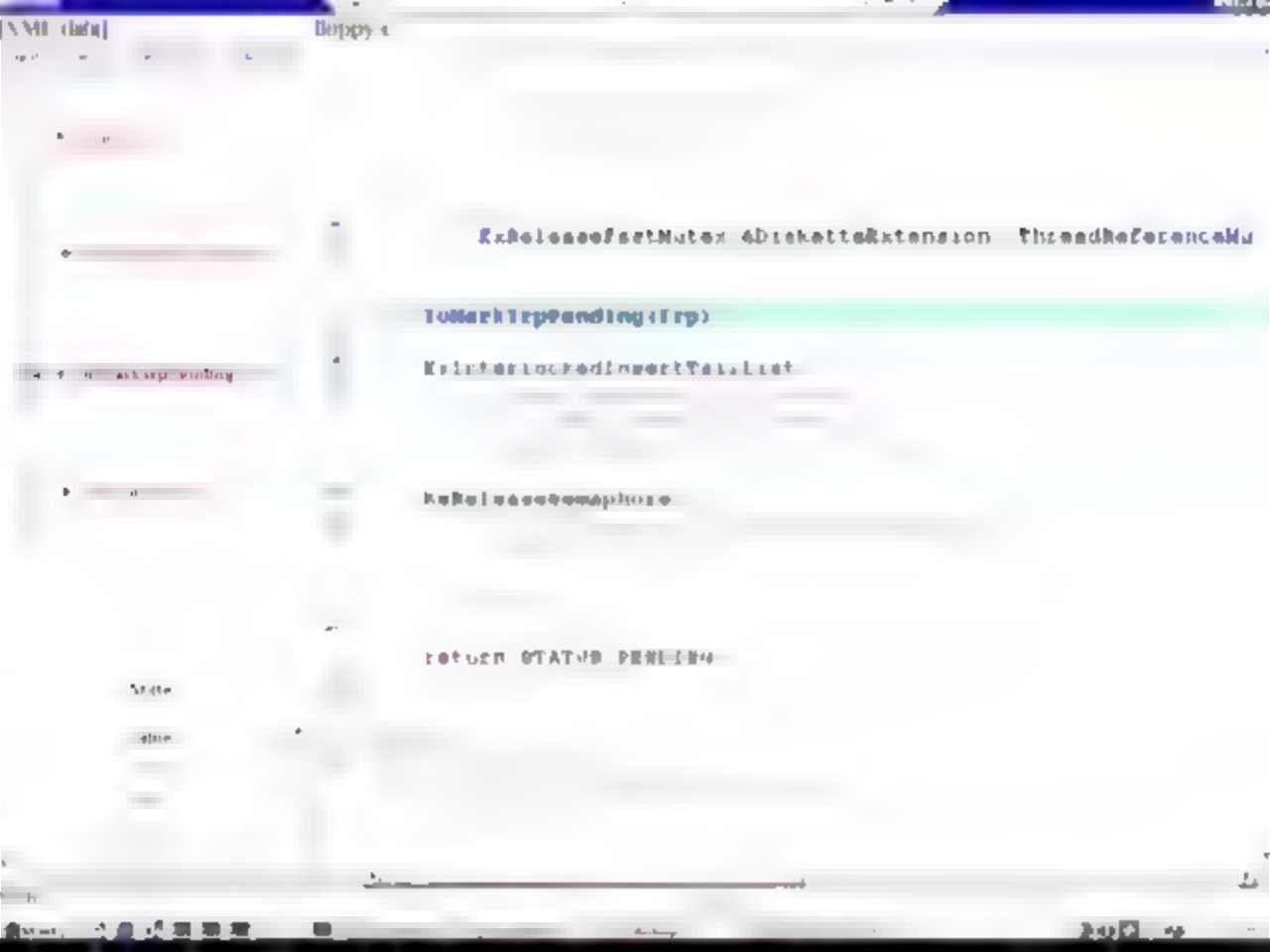


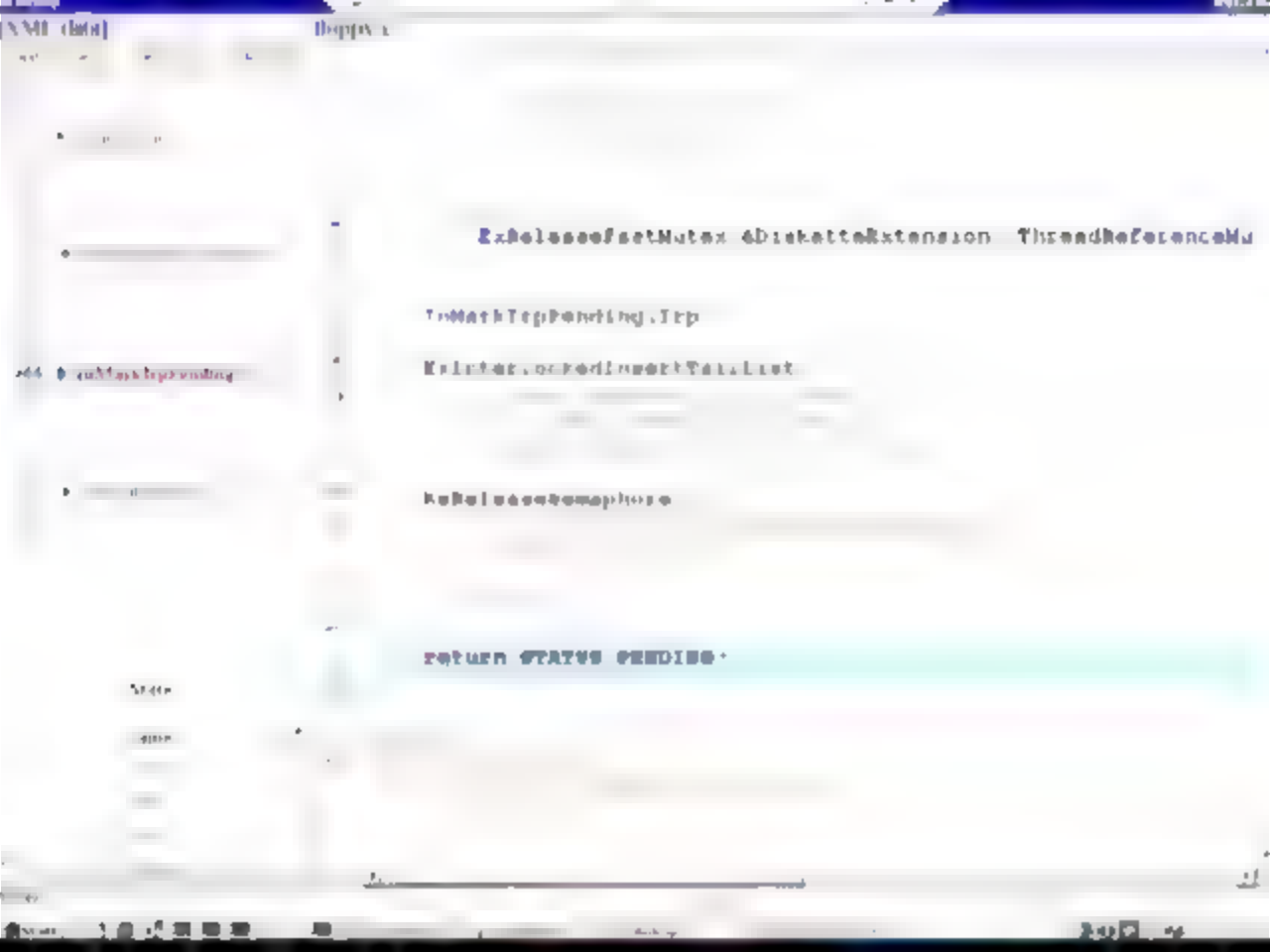


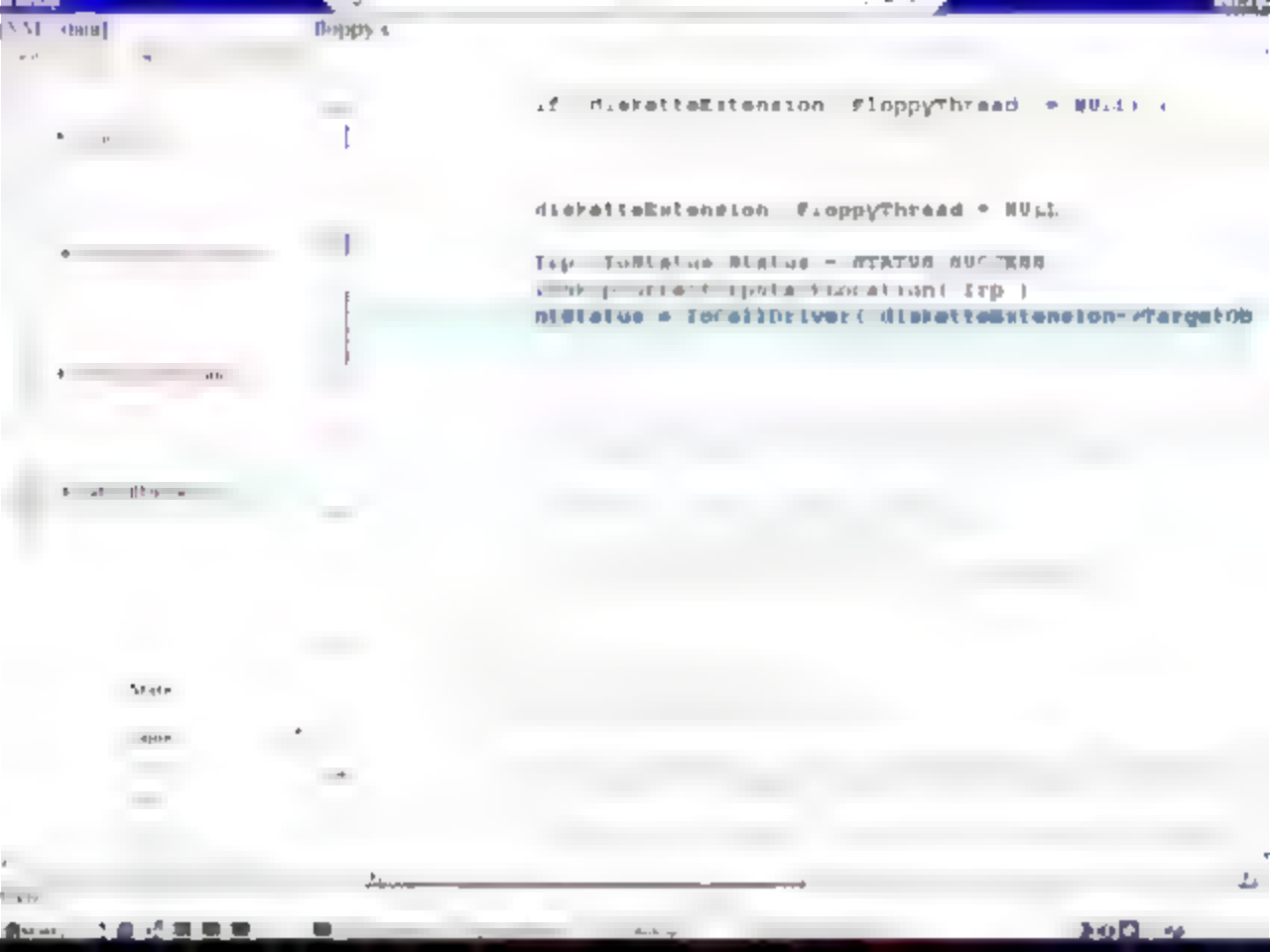


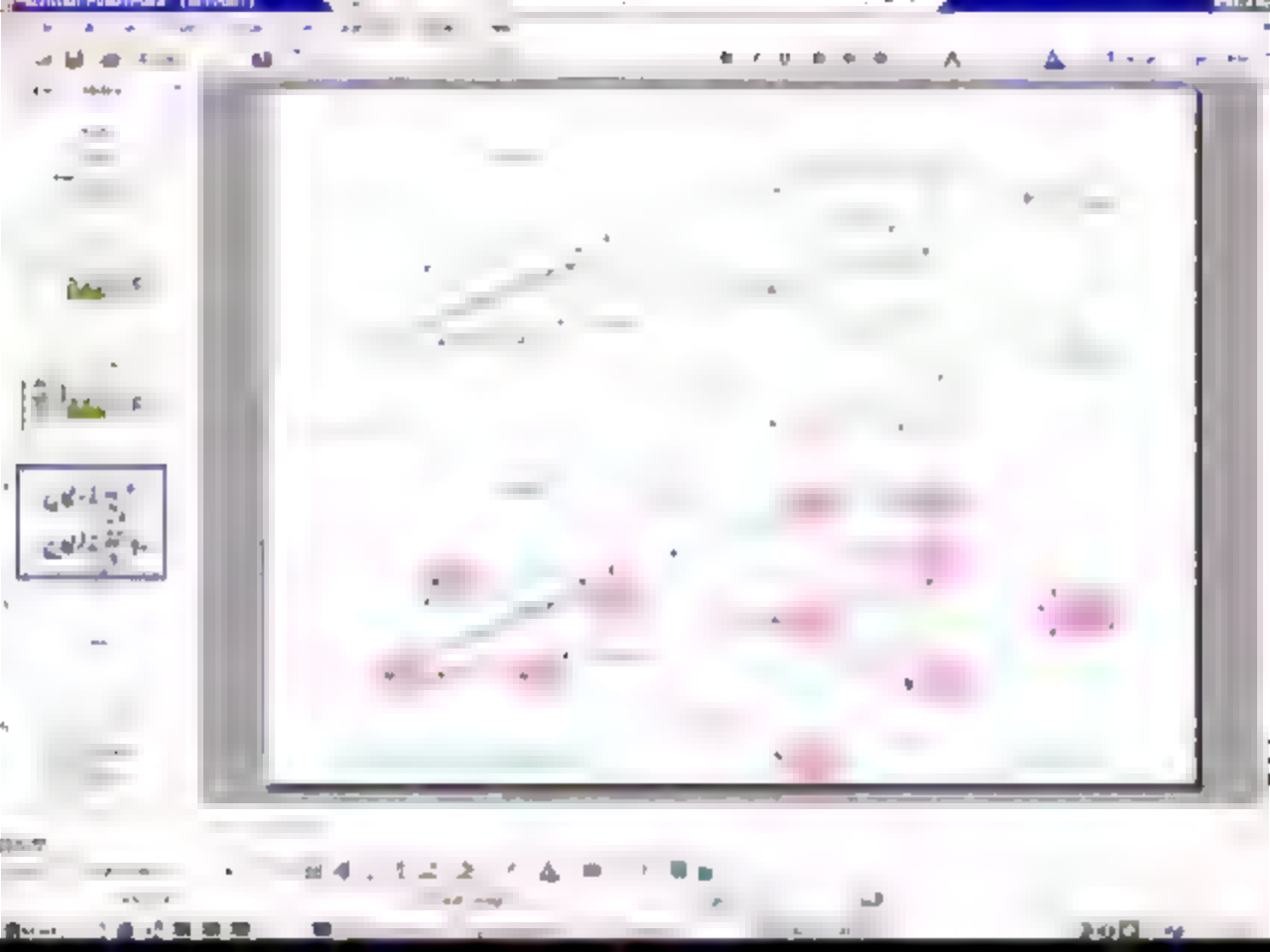


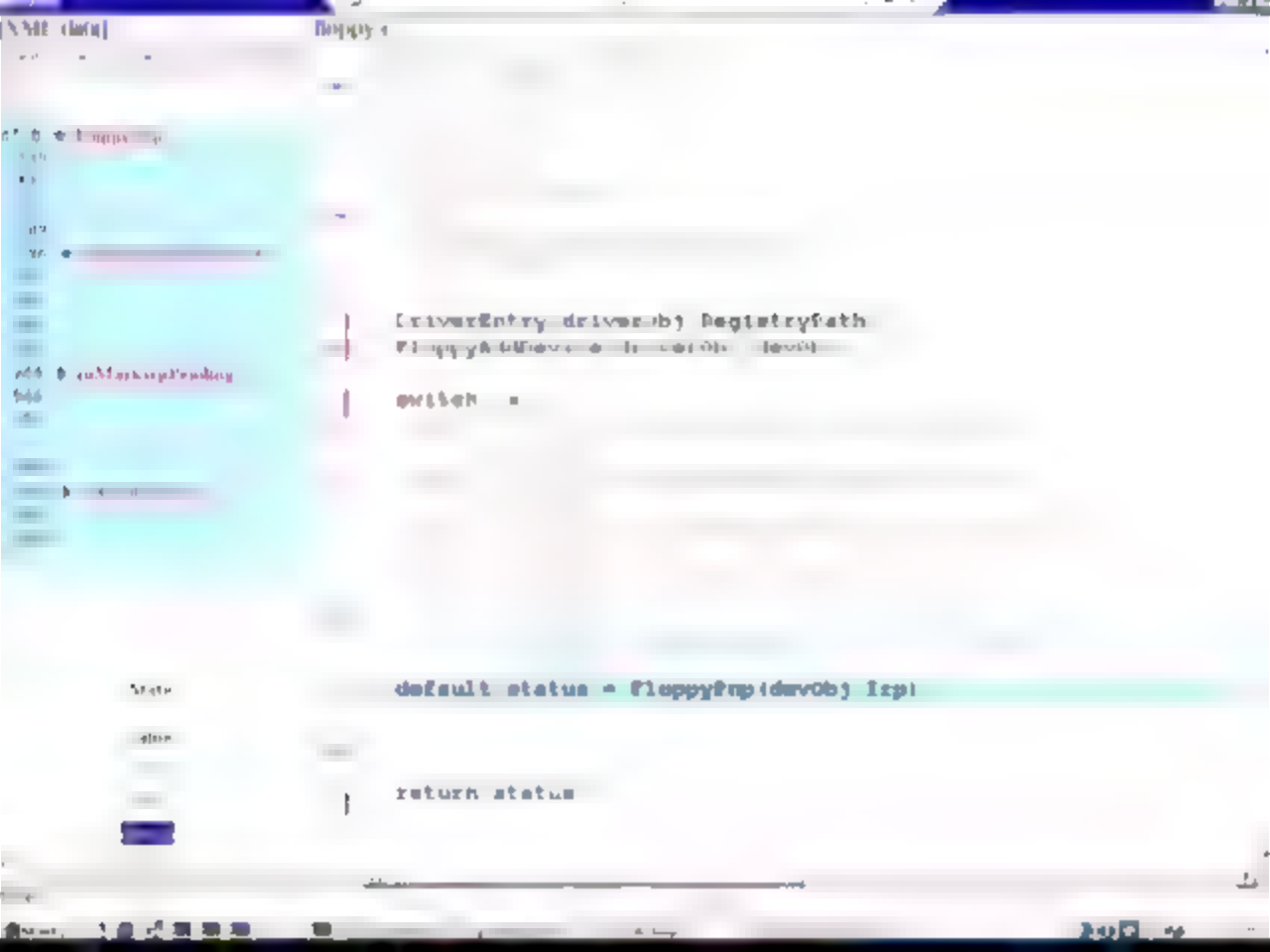










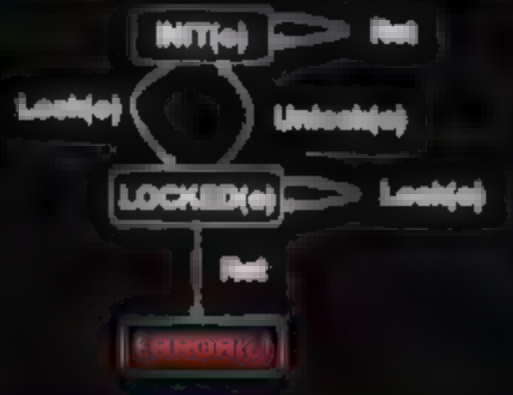


ESP (Manuvir Das)

- Error detection via scalable program analysis
- Similar to SLAM:
  - SLIC-like specification
  - Automatic analysis
  - Sound (finds all errors)
- Differences from SLAM:
  - Targeted at large (million+ LOC) C++ code bases
  - Only parameterized finite state protocols
  - Weaker (less precise) analysis

Parameterized Protocols

- FSM attached to data items
- Syntactic patterns trigger transitions



C++ code pattern	Transition
o->m_spinlock.Get()	Lock(o)
o->m_spinlock.Release()	Unlock(o)
return o	Ret

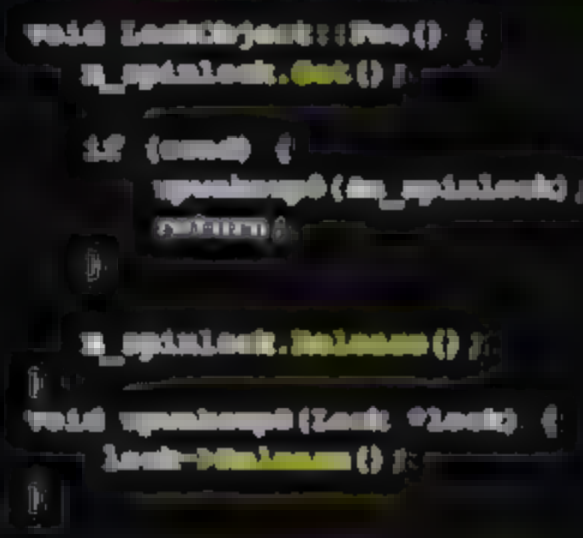
ESP Insight

- Three distinct entities
  - Sequence of actions along control flow path
  - Data involved in actions
    - o->Get(); p = o; p->Release();
  - Data involved in path feasibility
    - if (c) o->Get(); ... if (c) o->Release();
- Insight: use different static analyses to track each entity

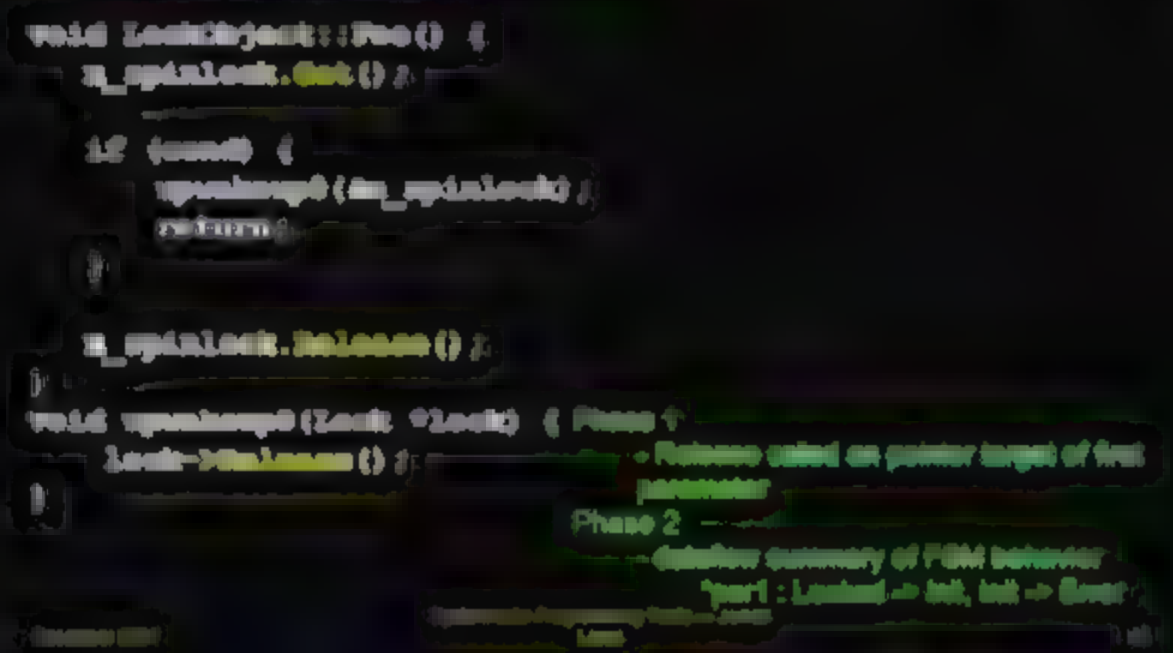
Three Phases of Static Analysis

1. Global value flow analysis
   - Use GOLF to build call and value flow graph
2. Global protocol tracking
   - Use dataflow and value flow to track protocols associated with data items
3. Local feasibility analysis
   - Use local abstract simulation + summaries from Phase II to rule out infeasible paths

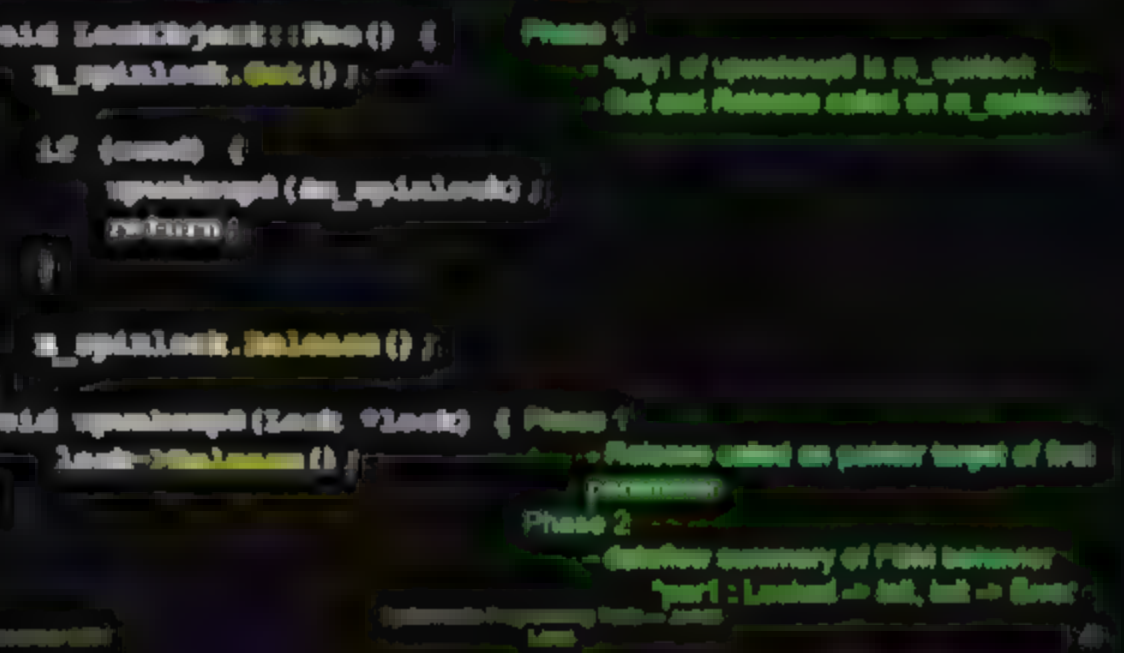
Example From SQL



Example From SQL

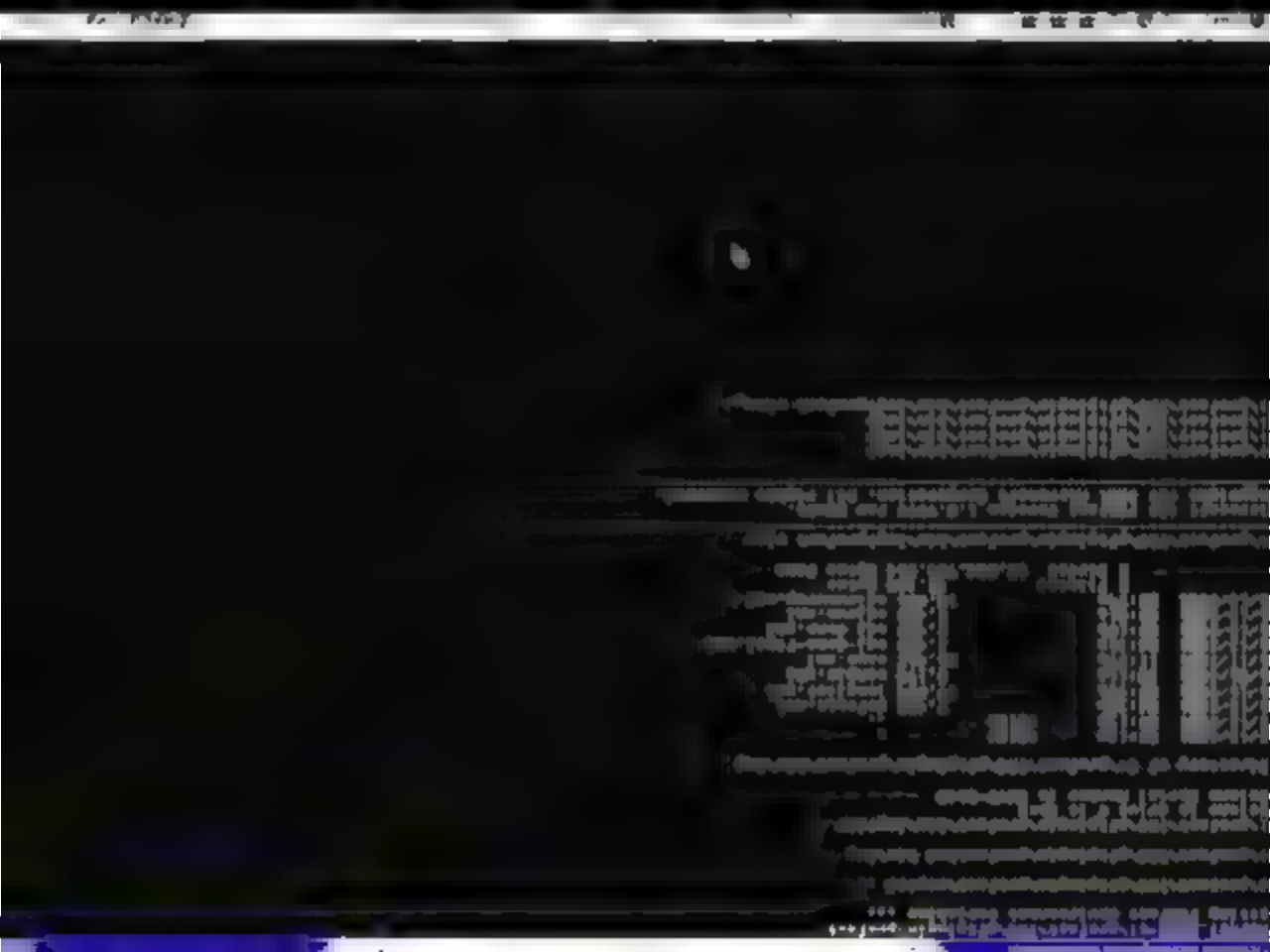


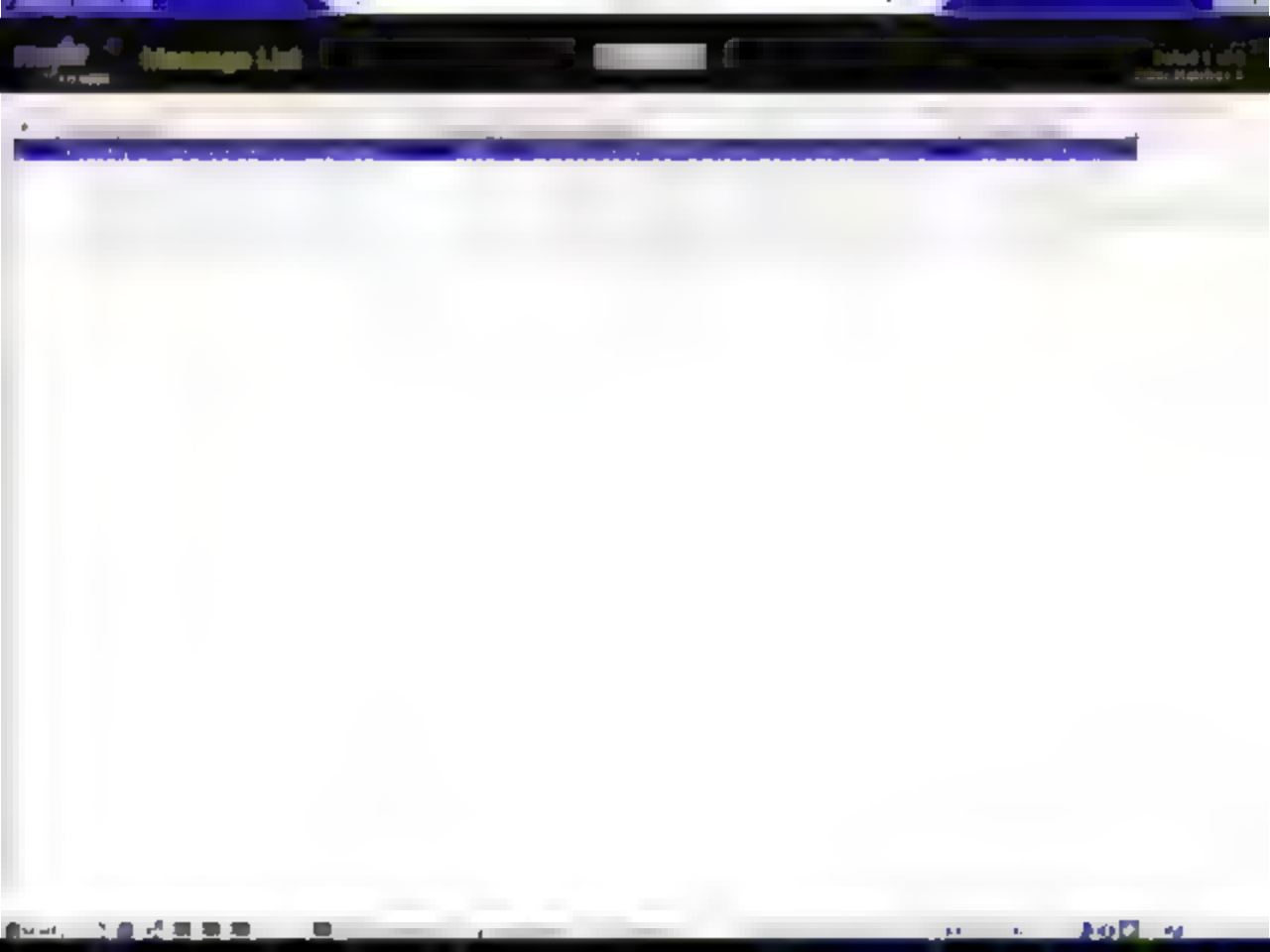
Example From SQL

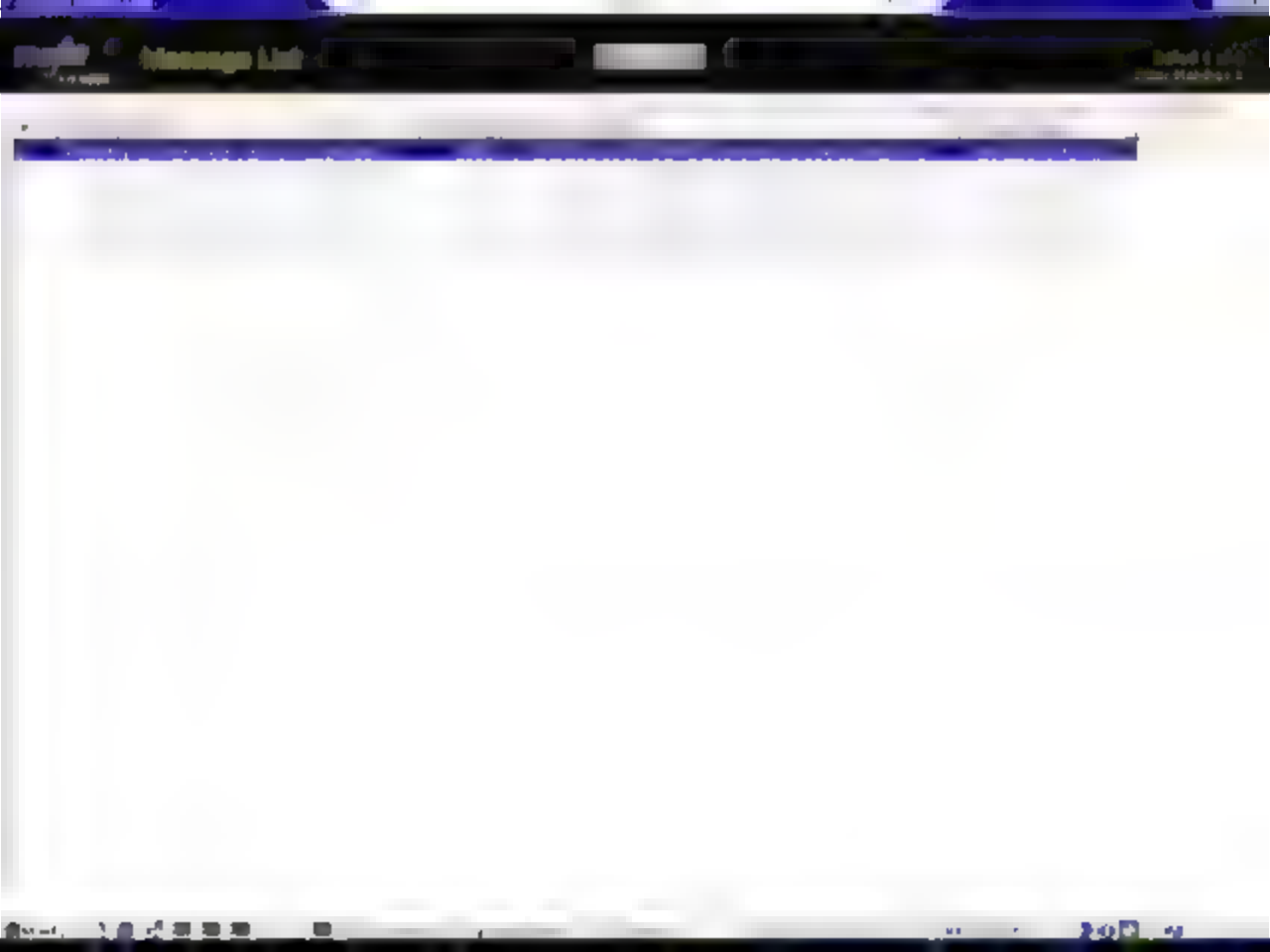


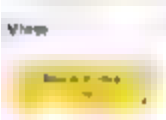
ESP Summary

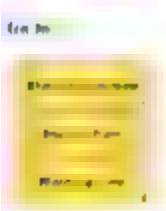
- Early implementation for C
  - Verified stdio usage (15 files) in 150K LOC program
- Extend to full C++ and improve speed
- Hypothesis
  - For many (protocols) errors, scalable analysis is sufficiently precise
  - Intricate usage patterns are difficult for programmers, as well as tools



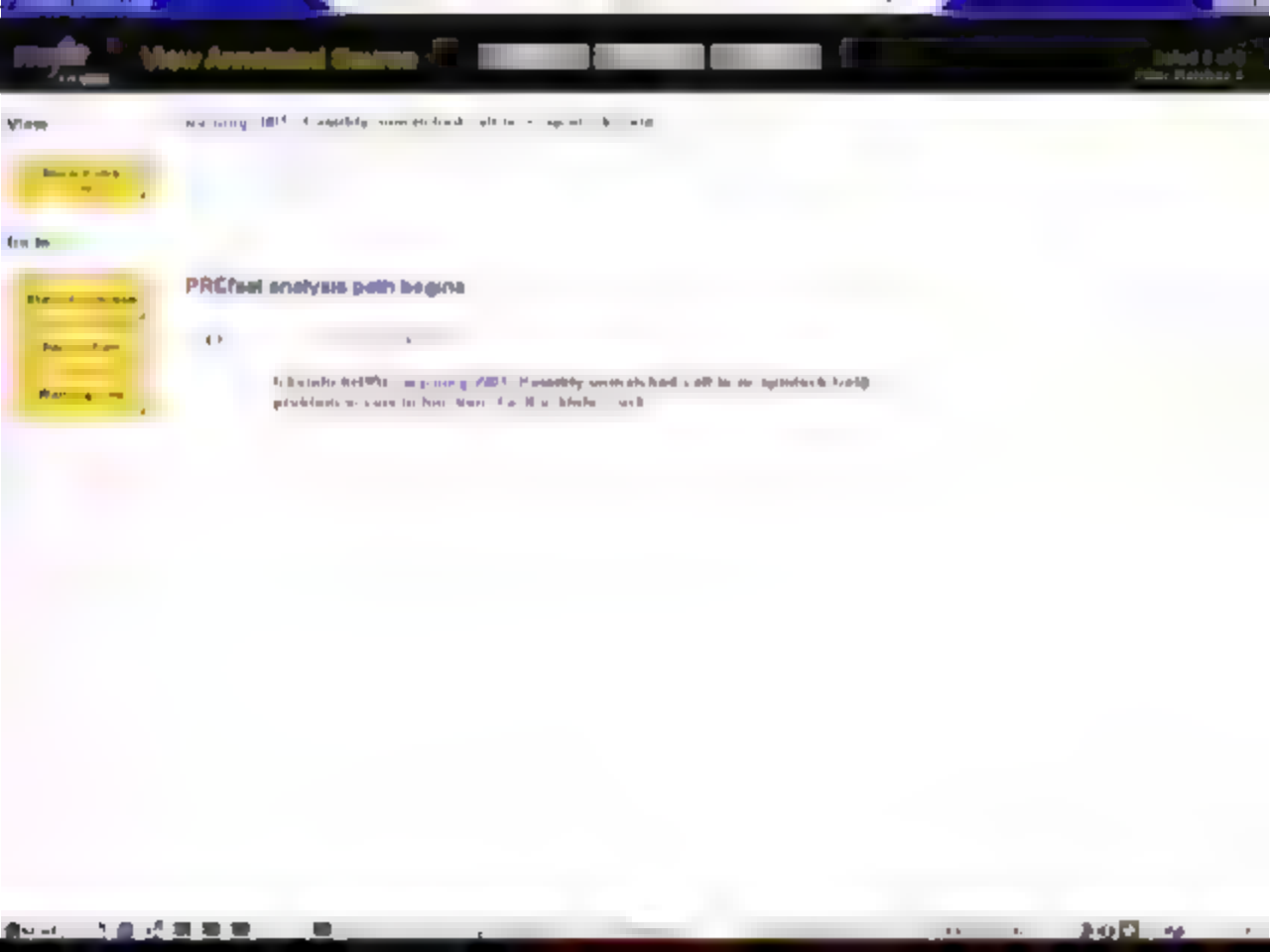


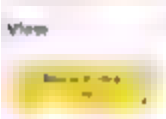


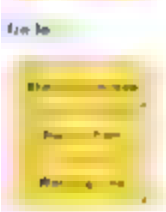




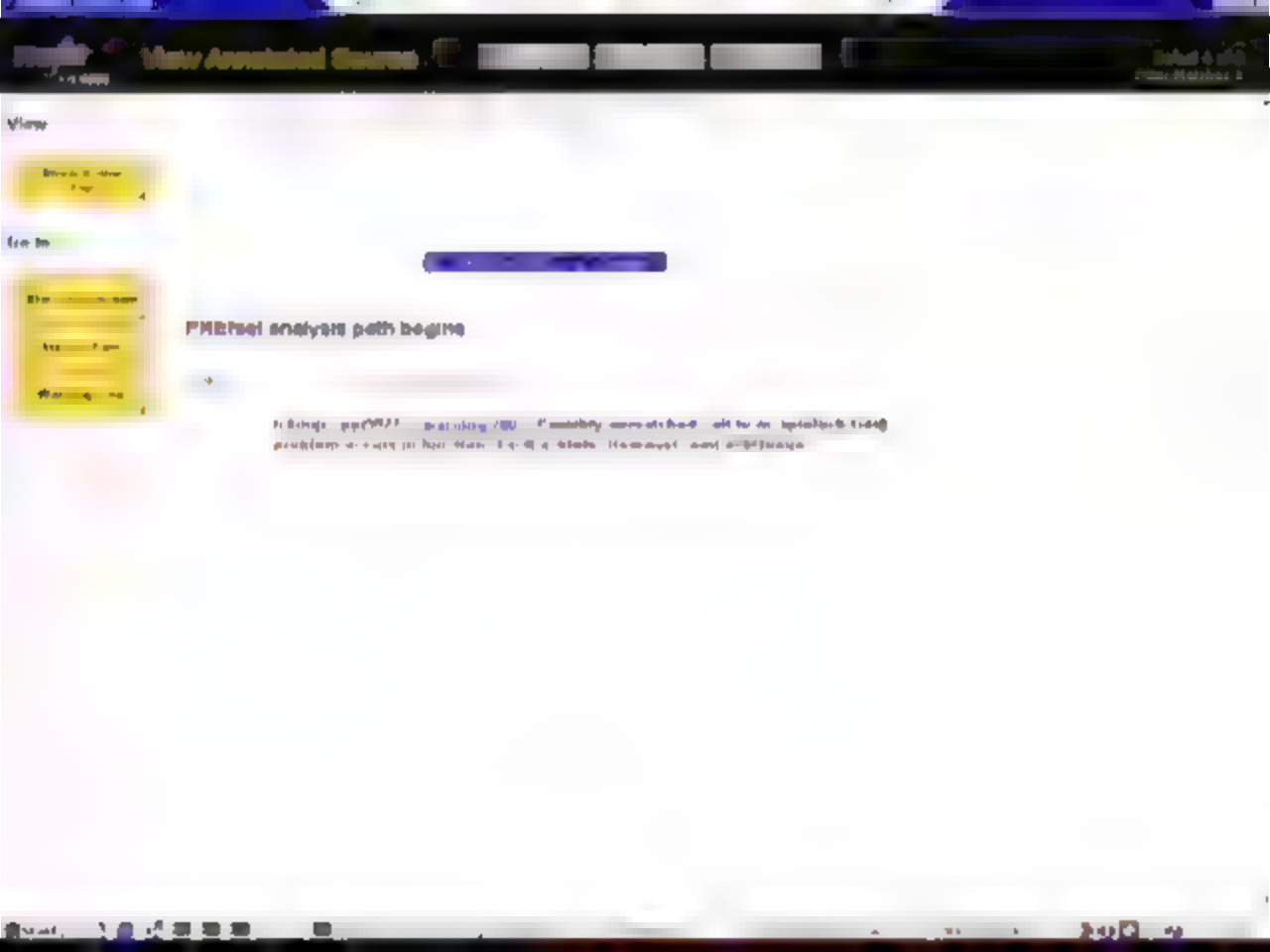
PAHfast analysis path begins

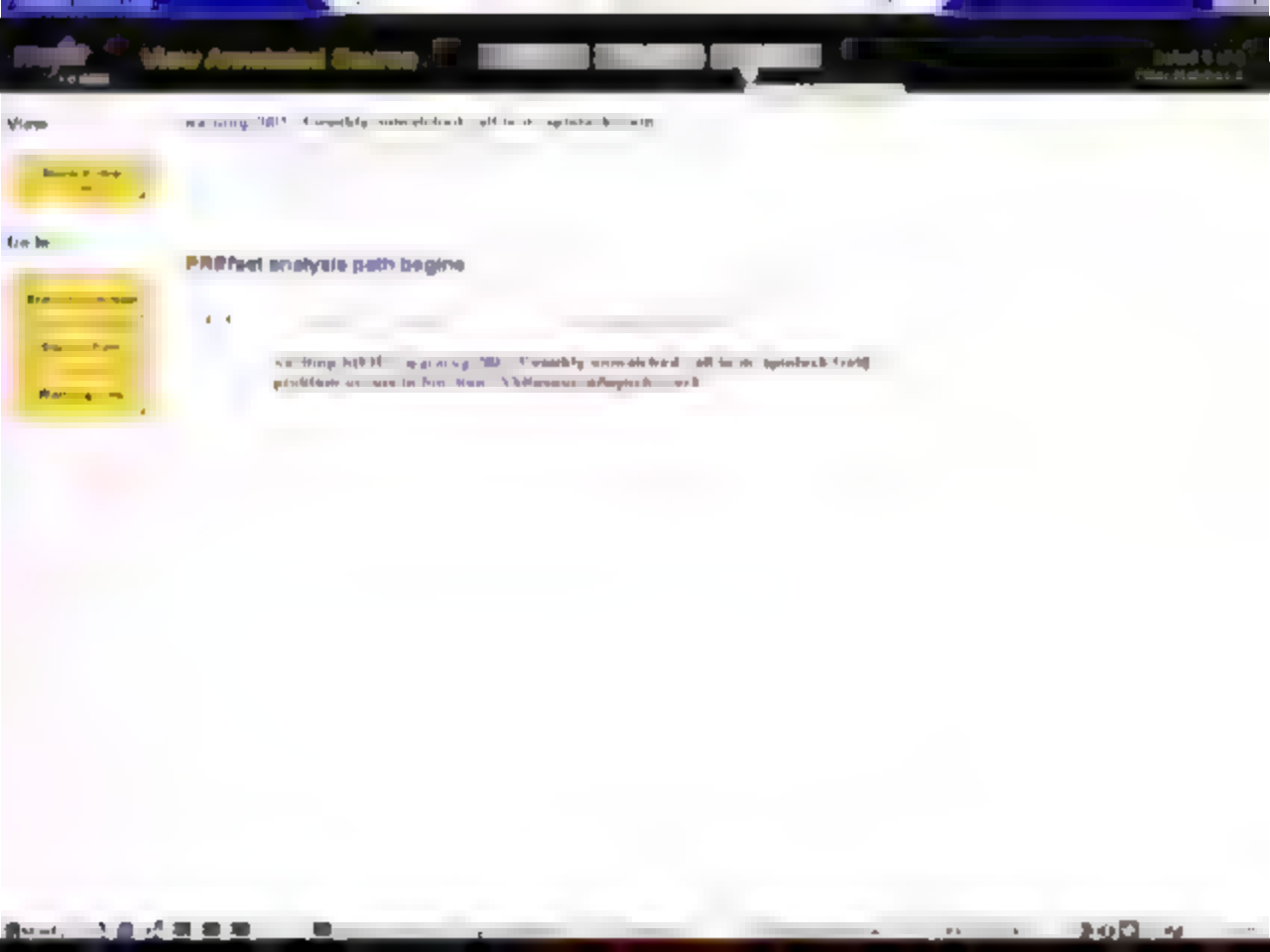


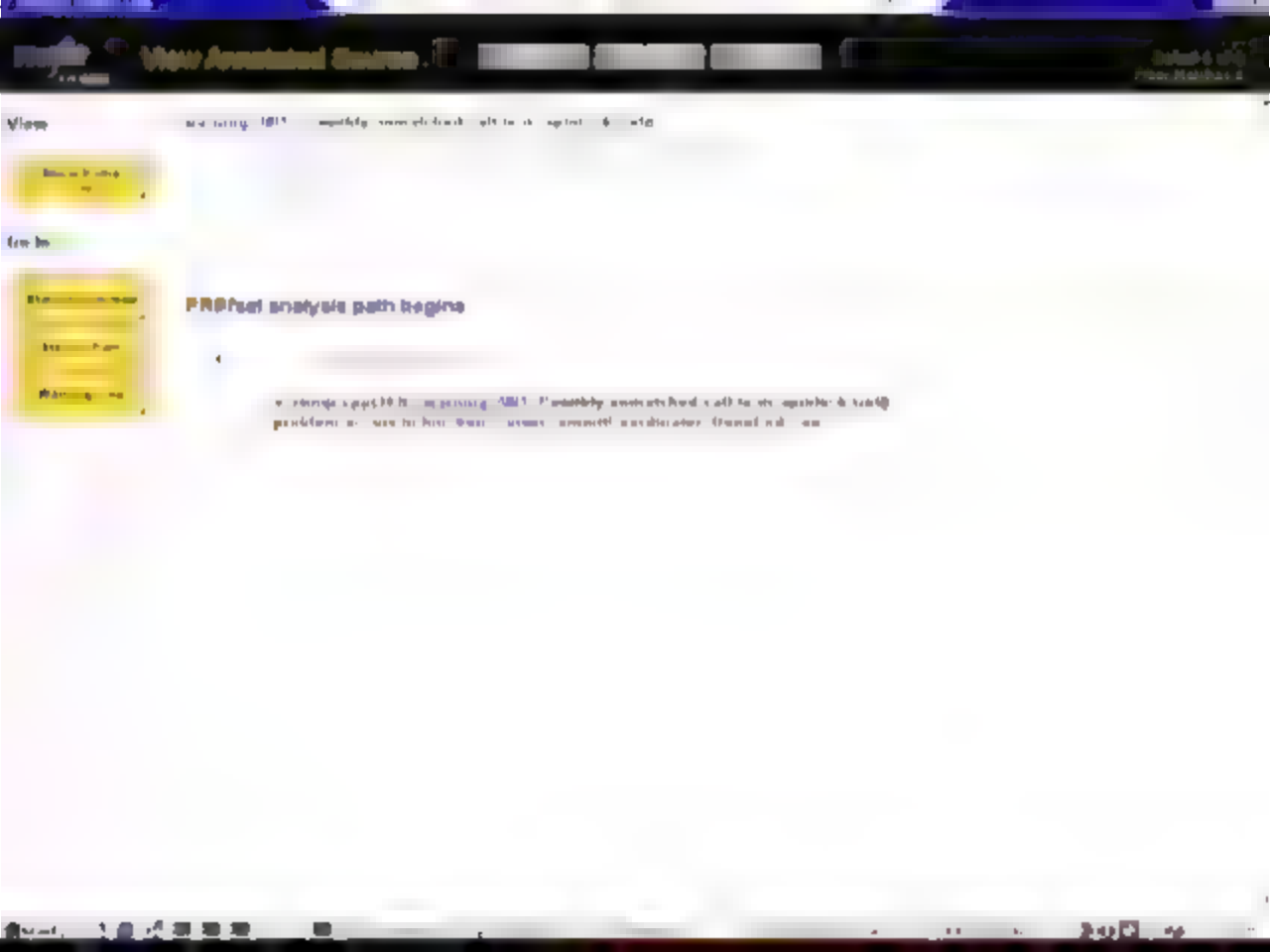


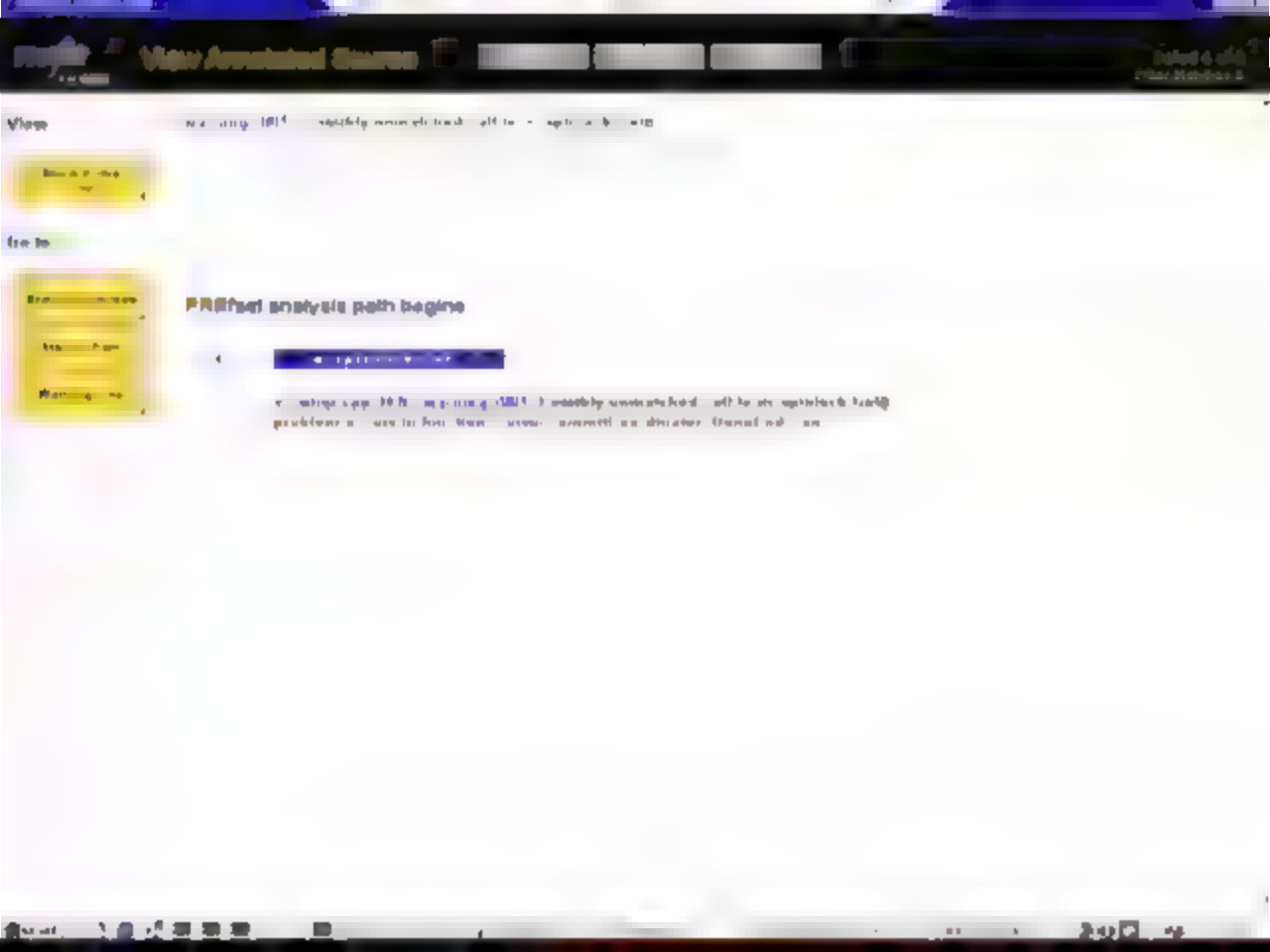


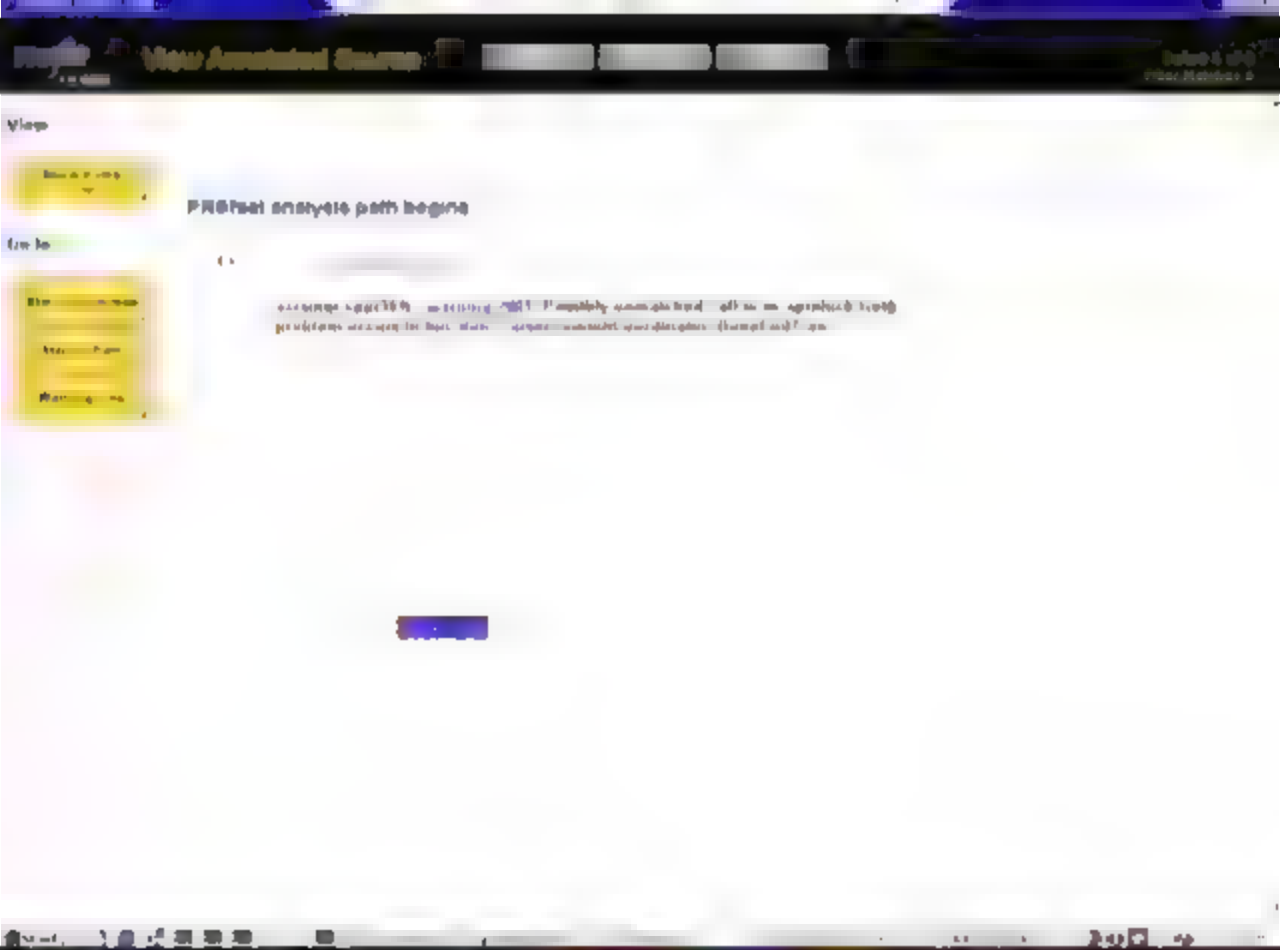
PREfast analysis path begins

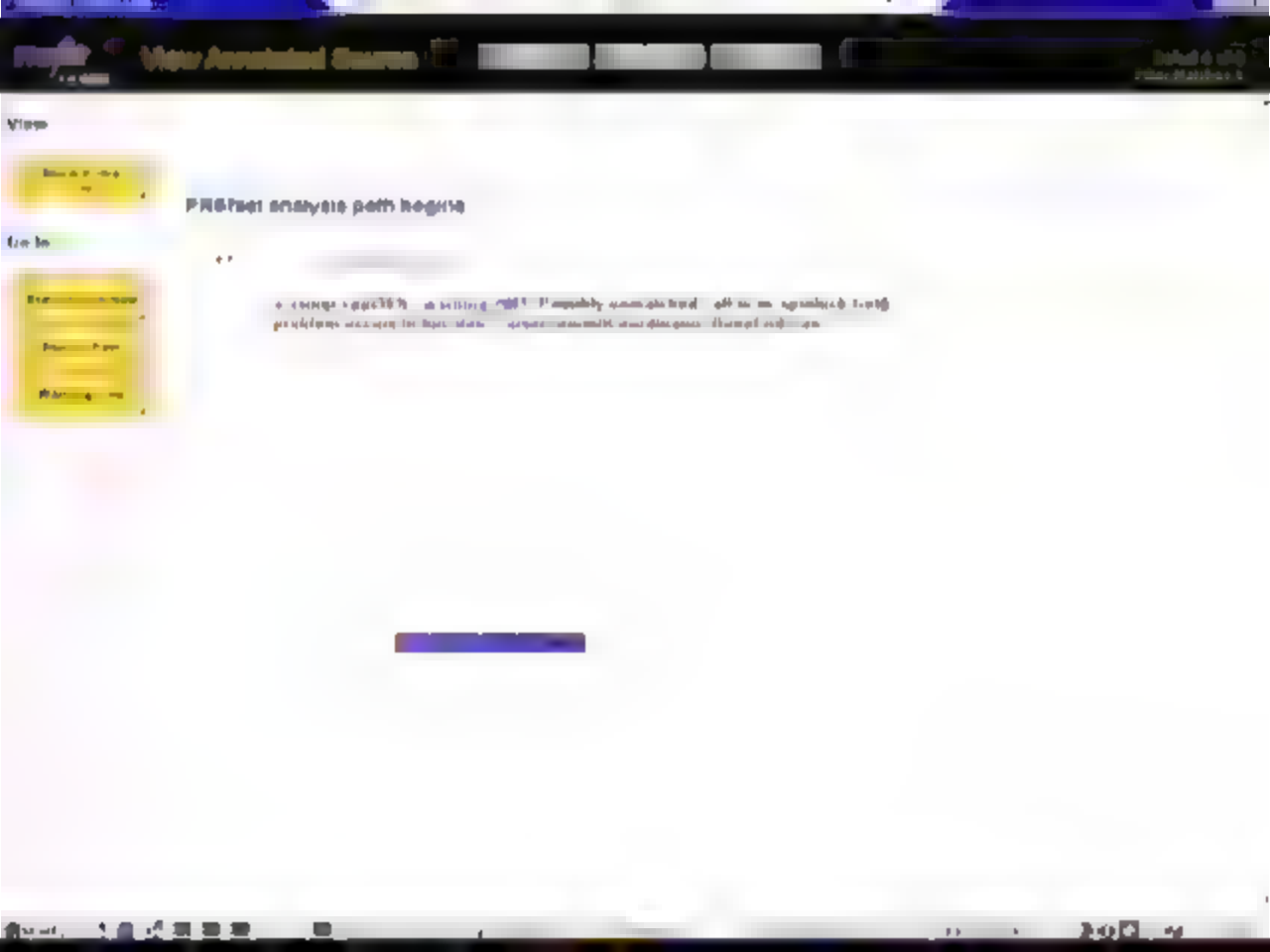


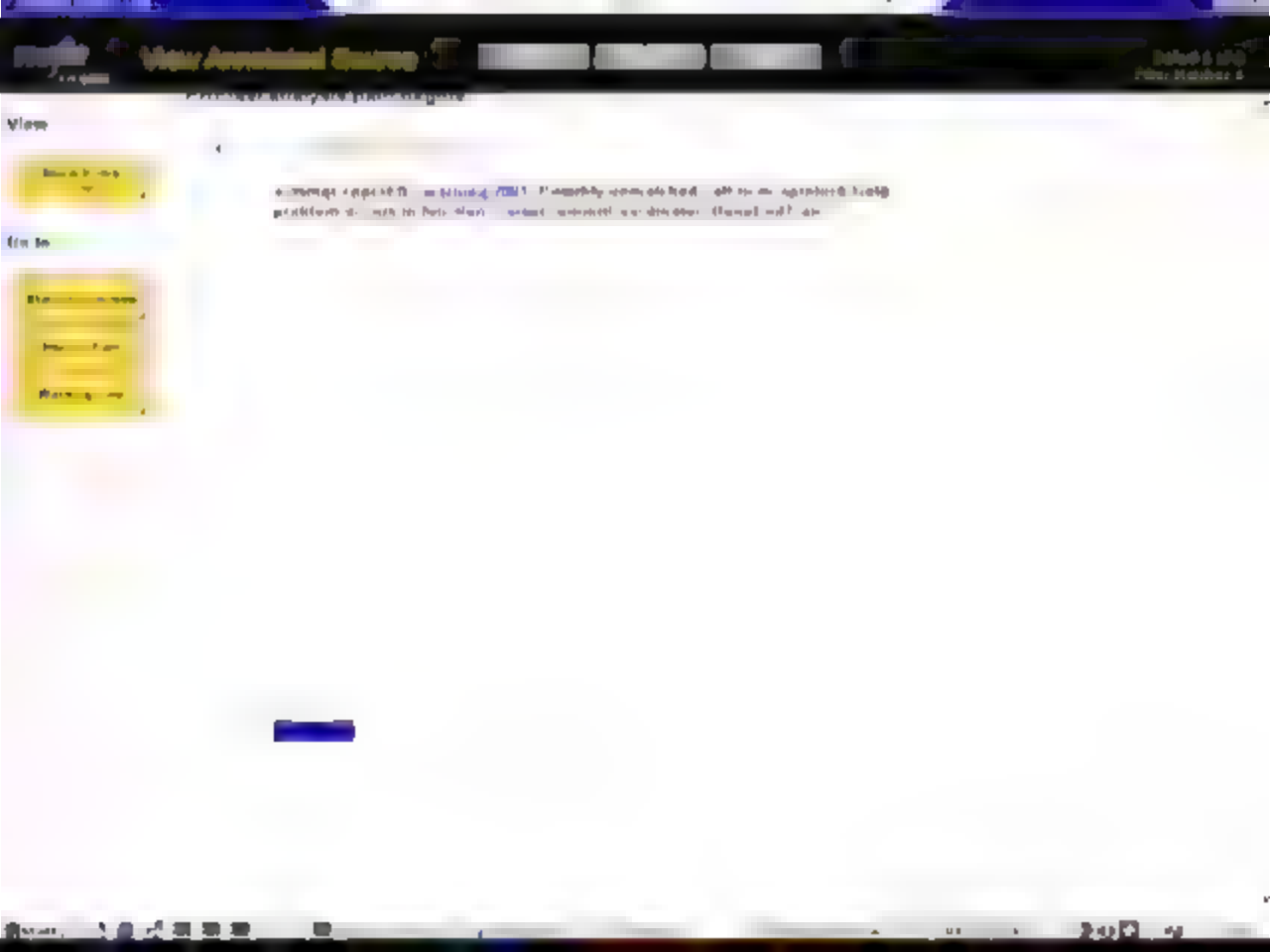












Concurrency

- Parallel and concurrent programs are hard to write, debug, test, modify, ...
  - Everyone is going to be writing many more of them
  - Cheap SMPs and .net
- Asynchronous programming
  - Message-passing (non-RPC) communication
  - Haskell++, Inega, ...
  - BizTalk
- Difficult behavioral properties
  - Deadlock-freedom
  - Communication progress
  - Message understood
- Few tools

Sharpie Example

```
// Declaration of client
operation client(in ch: x, out ch: w) @ x? => w!

// Implementation of server
server (in ch: z, out ch: y) @ z? => y!
{
  select z?() =>
    async c, d. client(c, d)
    in
      select d?() => y!
}
```

Sharpie Example

```
// Declaration of client
operation client(in ch: x, out ch: w) @ x? => w!

// Implementation of server
server (in ch: x, out ch: y) @ x? => y!
{
  select x?0 =>
    server.c.d.client(c, d)
  in
    select d?0 => y!
}
```

Sharpie Example

```
// Declaration of client
operation client(In ch: z, Out ch: w) @ z? -> w!

// Implementation of server
server (In ch: x, Out ch: y) @ x? -> y!
{
  select x?() ->
    async c, d, client(c, d)
    in
      select d?() -> y!
}
```

Receive on z
Send on w

Async call on client

Receive on d

Security Analysis

- Many security flaws exploit information flow

```
UserInput(s1);
...
CallToKernel(v);
```

- Value flow analysis
  - Determines where value comes from or goes to during execution
  - "Can the value of string s1 flow to variable v during execution?"
  - Previously, interprocedural analysis was imprecise or expensive

GOLF (Manuvir Das, Manuel Fahndrich, Jakob Rehof)

- Generalized One Level Flow
  - Whole-program value flow engine
- Scalable, context-sensitive, conservative analysis
  - Handles function calls accurately (context-sensitive)
  - Efficient (MLOC in seconds)
  - Precise (experimental evidence)
- Algorithms incorporated into PREfix
- Basis for family of security tools

SPT Summary

- SPT approach
  - Programmer/tester writes partial specifications
  - Tools ensure code follows specs
    - SLAM: extremely precise analysis
    - ESP: trade precision for scalability
    - Vault: integrate approach into language
    - Behave!: concurrency
    - GOLF: value flow problems
- Why so many projects?
  - Partial verification, not total correctness